JFP **24** (2-3): 384–418, 2014.
doi:10.1017/S095679681400001X First published online 31 January 2014

*Propositions as sessions**

PHILIP WADLER

University of Edinburgh, South Bridge, Edinburgh EH8 9YL, UK
(*e-mail:* wadler@inf.ed.ac.uk)

Abstract

Continuing a line of work by Abramsky (1994), Bellin and Scott (1994), and Caires and Pfenning (2010), among others, this paper presents CP, a calculus, in which propositions of classical linear logic correspond to session types. Continuing a line of work by Honda (1993), Honda *et al.* (1998), and Gay & Vasconcelos (2010), among others, this paper presents GV, a linear functional language with session types, and a translation from GV into CP. The translation formalises for the first time a connection between a standard presentation of session types and linear logic, and shows how a modification to the standard presentation yields a language free from races and deadlock, where race and deadlock freedom follows from the correspondence to linear logic.

> *'The new connectives of linear logic have obvious meanings in terms of parallel computation. [. . .] Linear logic is the first attempt to solve the problem of parallelism* at the logical level*, i.e., by making the success of the communication process only dependent of the fact that the programs can be viewed as* proofs *of something, and are therefore sound'.*
>
> —Girard 1987 (emphasis as in the original)

1 Introduction

Functional programmers know where they stand: upon the foundation of λ-calculus. Its canonicality is confirmed by its double discovery, once as natural deduction by Gentzen and again as λ-calculus by Church. These two formulations are related by the Curry–Howard correspondence, which takes

propositions *as* types,
proofs *as* programs, and
normalisation of proofs *as* evaluation of programs.

The correspondence arises repeatedly: Hindley's type inference corresponds to Milner's Algorithm W; Girard's System F, corresponds to Reynold's polymorphic λ-calculus; Peirce's law in classical logic corresponds to Landin's J operator (better known as call/cc).

* The online version of this paper uses colour to highlight the relation of types to terms and source to target.

Today, mobile phones, server farms, and many-core processors make us concurrent programmers. Where lies a foundation for concurrency as firm as that of λ-calculus? Many process calculi have emerged – ranging from Communicating Sequential Processes (CSP) to Calculus of Communicating Systems (CCS) to π-calculus to join calculus to mobile ambients to bigraphs – but none is as canonical as λ-calculus, and none has the distinction of arising from Curry–Howard. This paper takes a further step along the programme pursued by Girard (1987), Abramsky (1994), Honda (1993), Caires & Pfenning (2010), among others, of seeking foundations for concurrency that rest upon the correspondence uncovered by Curry and Howard.

Since its inception by Girard (1987), linear logic has held the promise of a foundation for concurrency rooted in Curry–Howard. In an early step, Abramsky (1994) and Bellin & Scott (1994) devised a translation from linear logic into π-calculus. Along another line, Honda (1993) introduced session types, further developed by Honda *et al.* (1998) and others, which take inspiration from linear logic, but do not enjoy a relationship as tight as Curry–Howard.

Recently, Caires & Pfenning (2010) found a twist on Abramsky's (1994) translation that yields an interpretation strongly reminiscent of session types, and a variant of Curry–Howard with

propositions *as* session types,
proofs *as* processes, and
cut elimination *as* communication.

The correspondence is developed in a series of papers by Caires, Pfenning, Toninho, and Pérez. This paper extends these lines of work with the following three contributions.

Firstly, inspired by the calculus πDILL of Caires & Pfenning (2010), this paper presents the calculus CP. Based on dual intuitionistic linear logic, πDILL uses two-sided sequents, with two constructs corresponding to output ($\otimes$ on the right of a sequent and $\multimap$ on the left), and two constructs corresponding to input ($\multimap$ on the right of a sequent and $\otimes$ on the left). Based on classical linear logic, CP uses one-sided sequents, offering greater simplicity and symmetry, with a single construct for output ($\otimes$) and a single construct for input ($\parr$), each dual to the other. Caires *et al.* (2012a) compare πDILL with πCLL, which like CP is based on classical linear logic; we discuss this comparison in Section 5. (If you like, CP stands for Classical Processes.)

Secondly, although πDILL is clearly reminiscent of the body of work on session types, no one has previously published a formal connection. Inspired by the linear functional language with session types of Gay & Vasconcelos (2010), this paper presents the calculus GV, and also presents a translation from GV into CP, for the first time formalising a tight connection between a standard presentation of session types and a standard presentation of linear logic. In order to facilitate the translation, GV differs from the language of Gay & Vasconcelos (2010) in some particulars. These differences suffice to make GV, unlike the original, free from races and deadlock. (If you like, GV stands for Good Variation.)

Curry–Howard relate proof normalisation to computation. Logicians devised proof normalisation to show consistency of logic, and for this purpose it is essential that proof

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

normalisation terminates. Hence, a consequence of Curry–Howard is that it identifies a fragment of λ-calculus for which the Halting Problem is solved. Well-typed programs terminate unless they explicitly resort to non-logical features such as general recursion. Similarly, a consequence of the Curry–Howard for concurrency is that it identifies a fragment of a process calculus which is free of deadlock. In particular, both πDILL and CP are such fragments, and the proof that GV is deadlock-free follows immediately from its translation to CP. We return to the question of what non-logical features might restore races and deadlock in the Conclusions.

Thirdly, this paper presents a calculus with a stronger connection to linear logic, at the cost of a weaker connection to traditional process calculi. Bellin & Scott (1994) and Caires & Pfenning (2010) each present a translation from linear logic into π-calculus such that cut elimination converts one proof to another if and only if the translation of the one reduces to the translation of the other; but to achieve this tight connection several adjustments are necessary.

Bellin & Scott (1994) restrict the axiom to atomic type, and Caires & Pfenning (2010) omit the axiom entirely. In terms of a practical programming language, such restrictions are excessive. The former permit type variables, but instantiating a type variable to a type requires restructuring the program (as opposed to simple substitution); the latter outlaw type variables altogether. Consequently, neither system lends itself to parametric polymorphism.

Further, Bellin & Scott (1994) only obtain a tight correspondence between cut elimination and π-calculus for the multiplicative connectives, and they require a variant of π-calculus with surprising structural equivalences such as $x(y).x(z).P \equiv x(z).x(y).P$ – permuting two reads on the same channel! Caires & Pfenning (2010) only obtain a tight correspondence between cut elimination and π-calculus by ignoring commuting conversions; this is hard to justify logically because commuting conversions play an essential role in cut elimination. Pérez *et al.* (2012) show commuting conversions correspond to contextual equivalences, but fail to capture the directionality of the commuting conversions.

Thus, while the connection established in previous work between cut elimination in linear logic and reduction in π-calculus is encouraging, it comes at a cost. Accordingly, this paper cuts the Gordian knot: It takes the traditional rules of cut elimination as specifying the reduction rules for its process calculus. Pro: We let logic guide the design of the 'right' process calculus. Con: We forego the assurance that comes from double discoveries of the same system, as with Gentzen & Church, Hindley & Milner, Girard (1987), Reynolds, and Peirce & Landin. Mitigating the con are the results cited above that show a connection between Girard's (1987) linear logic and Milner's π-calculus, albeit not as tight as the other connections just mentioned.

In return for loosening its connection to π-calculus, the design of CP avoids the problems described above. The axiom is interpreted at all types, using a construct suggested by Caires *et al.* (2012a), and consequently it is easy to extend the system to support polymorphism, using a construct suggested by Turner (1995). All commuting conversions of cut elimination are satisfied.

This paper is the journal version of Wadler (2012). It is organised as follows. Section 2 sketches the path from Abramsky, Bellin, Scott, Caires, and Pfenning to the present one.

Section 3 presents CP. Section 4 presents GV and its translation to CP. Section 5 discusses related works. Section 6 concludes.

2 The twist

This section explains how a small but crucial twist relates the work of Abramsky, Bellin, and Scott to the work of Caires and Pfenning and to what is presented here.

The key difference is in the interpretation of the linear connectives $\otimes$ and $\parr$. In contrast, all of these works agree in their interpretation of $\oplus$ and $\&$ as making a selection and offering a choice, and indeed Honda (1993) already uses $\oplus$ and $\&$ in that way. But input and output in Honda's (1993) work appear to have no connection with the connectives of linear logic. (It is an unfortunate historical accident that ! and ? denote output and input in many process calculi, including those of Honda (1993) and Gay & Vasconcelos (2010), while ! and ? denote exponentials in linear logic; the two uses are distinct and should not be confused.)

In Abramsky (1994) and Bellin & Scott (1994), the following two rules interpret the linear connectives $\otimes$ and $\parr$.

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, z : B}{\nu y, z.x\langle y, z\rangle.(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B}\ \otimes \qquad \frac{R \vdash \Theta, y : A, z : B}{x(y, z).R \vdash \Theta, x : A \parr B}\ \parr$$

Under their interpretation, $A \otimes B$ is the type of a channel which outputs a pair of an A and a B, while $A \parr B$ is the type of a channel which inputs a pair of an A and a B. In the rule for $\otimes$, process $\nu y, z.x\langle y, z\rangle.(P \mid Q)$ allocates fresh channels y and z, transmits the pair of channels y and z along x, and then concurrently executes P and Q. In the rule for $\parr$, process $x(y, z).R$ communicates along channel x obeying protocol $A \parr B$; it receives from x the pair of names y and z, and then executes R.

This work puts a twist on the above interpretation. Here we use the following two rules to interpret the linear connectives $\otimes$ and $\parr$.

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{\nu y.x\langle y\rangle.(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B}\ \otimes \qquad \frac{R \vdash \Theta, y : A, x : B}{x(y).R \vdash \Theta, x : A \parr B}\ \parr$$

Under the new interpretation, $A \otimes B$ is the type of a channel which outputs an A and then behaves as a B, while $A \parr B$ is the type of a channel which inputs an A and then behaves as a B. In the rule for $\otimes$, process $\nu y.x\langle y\rangle.(P \mid Q)$ allocates fresh variable y, transmits y along x, and then concurrently executes P and Q. In the rule for $\parr$, process $x(y).R$ receives name y along x, and then executes R.

The difference is that in the rules used by Abramsky (1994) and Bellin & Scott (1994) the hypotheses refer to channels y and z and the conclusion to channel x, while in the rules used here the hypotheses refer to channels y and x, and the conclusion reuses the channel x. One may regard the type of the channel as evolving as communication proceeds, corresponding to the notion of the session type.

While it is natural to interpret $A \otimes B$ and $A \parr B$ as transmitting and accepting a pair, it may initially seem unnatural to interpret $A \otimes B$ and $A \parr B$ asymmetrically, as first transmitting or accepting a channel obeying protocol A and then obeying protocol B. The

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

unnaturality of the interpretation may explain why it took 16 years between when Abramsky (1994) and Bellin & Scott (1994) published their interpretation of the linear connectives as pairing and Caires & Pfenning (2010) published their interpretation of the linear connectives as session types. In fact, we will see that there is an isomorphism between $A \otimes B$ and $B \otimes A$, and similarly between $A \parr B$ and $B \parr A$, which provides the necessary symmetry.

The insight behind this twist is clearly due to Caires & Pfenning (2010), although the relationship to the earlier work is not presented in their paper along the lines described. Indeed, the relation to the work of Abramsky (1994) and Bellin & Scott (1994) is further obscured because Caires & Pfenning (2010) use intuitionistic linear logic rather than classical logic.

In Caires & Pfenning (2010), the following four rules interpret the linear connectives $\otimes$ and $\multimap$.

$$\frac{\Gamma;\, \Delta \vdash P :: y : A \qquad \Gamma;\, \Delta' \vdash Q :: x : B}{\Gamma;\, \Delta, \Delta' \vdash \nu y.x\langle y\rangle.(P \mid Q) :: x : A \otimes B}\ \otimes\text{-R} \qquad \frac{\Gamma;\, \Delta, y : A, x : B \vdash R :: z : C}{\Gamma;\, \Delta, x : A \otimes B \vdash x(y).R :: z : C}\ \otimes\text{-L}$$

$$\frac{\Gamma;\, \Delta, y : A \vdash R :: x : B}{\Gamma;\, \Delta \vdash x(y).R :: x : A \multimap B}\ \multimap\text{-R} \qquad \frac{\Gamma;\, \Delta \vdash P :: y : A \quad \Gamma;\, \Delta', x : B \vdash Q :: z : C}{\Gamma;\, \Delta, \Delta', x : A \multimap B \vdash \nu y.x\langle y\rangle.(P \mid Q) :: z : C}\ \multimap\text{-L}$$

To print these rules requires more than twice as much ink as to print the comparable rules used here! Part of the difference is due to different forms of bookkeeping, which is mostly incidental and won't be detailed here. Another difference is that the above rules use $\multimap$ instead of $\parr$, but since $A \multimap B$ and $A^{\perp} \parr B$ are equivalent, this is not so significant. The difference we will focus on here is that the use of an intuitionistic logic forces Caires & Pfenning (2010) to represent output by two rules, $\otimes$-R and $\multimap$-L, and input by two rules, $\multimap$-R and $\otimes$-L. This duplication adds complexity and confusion. Worse, it impedes usability, since if one user defines an output protocol with $A \otimes B$, and the second user defines an input protocol with $A \multimap B$, then these cannot be connected directly with a cut, but require some form of mediating code. Avoiding mismatch requires some convention – for instance, a server might always use R rules and a client always use L rules. The classical approach is easier to use in practice, since no convention for servers and clients is required. Caires & Pfenning (2010) do give some reasons to prefer the intuitionistic approach to the classical one, and these are described in Section 5.

While Girard (1987) associated linear logic with concurrency from the beginning, many readers of the original paper (including the current author) had some difficulty giving an intuitive reading to the classical connective $\parr$. One advantage of the work presented here is that it may finally offer an intuitive reading of this fundamental connective.

3 Classical linear logic as a process calculus

This section presents CP, a session-typed process calculus. CP is based on classical linear logic with one-sided sequents, the system treated in the first paper on linear logic by Girard (1987).

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Types. Propositions, which may be interpreted as session types, are defined by the following grammar:

$$
\begin{array}{ll}
A,B,C ::= & \\
\quad X & \text{propositional variable} \\
\quad X^{\perp} & \text{dual of propositional variable} \\
\quad A \otimes B & \text{‘times’, output } A \text{ then behave as } B \\
\quad A \parr B & \text{‘par’, input } A \text{ then behave as } B \\
\quad A \oplus B & \text{‘plus’, select from } A \text{ or } B \\
\quad A \mathbin{\&} B & \text{‘with’, offer choice of } A \text{ or } B \\
\quad !A & \text{‘of course!’, server accept} \\
\quad ?A & \text{‘why not?’, client request} \\
\quad \exists X.B & \text{existential, output a type} \\
\quad \forall X.B & \text{universal, input a type} \\
\quad 1 & \text{unit for } \otimes \\
\quad \perp & \text{unit for } \parr \\
\quad 0 & \text{unit for } \oplus \\
\quad \top & \text{unit for } \&
\end{array}
$$

Let A,B,C range over propositions, and X,Y,Z range over propositional variables. Every propositional variable X has a dual written $X^{\perp}$. Propositions are composed from multiplicatives ($\otimes,\parr$), additives ($\oplus$,&), exponentials ($!,?$), second-order quantifiers ($\exists,\forall$), and units ($1,\perp,0,\top$). In $\exists X.B$ and $\forall X.B$, propositional variable X is bound in B. Write $\mathsf{fv}(A)$ for the free variables in proposition A. Our notation is identical to that of Girard (1987), save we write quantifiers as $\exists,\forall$ where he writes $\vee,\wedge$.

Duals. Duals play a key role, ensuring that a request for input at one end of a channel matches an offer of a corresponding output at the other, and that a request to make a selection at one end matches an offer of a corresponding choice at the other.

Each proposition A has a dual $A^{\perp}$, defined as follows:

$$
\begin{array}{rlrl}
(X)^{\perp} &= X^{\perp} & (X^{\perp})^{\perp} &= X \\
(A \otimes B)^{\perp} &= A^{\perp} \parr B^{\perp} & (A \parr B)^{\perp} &= A^{\perp} \otimes B^{\perp} \\
(A \oplus B)^{\perp} &= A^{\perp} \mathbin{\&} B^{\perp} & (A \mathbin{\&} B)^{\perp} &= A^{\perp} \oplus B^{\perp} \\
(!A)^{\perp} &= ?A^{\perp} & (?A)^{\perp} &= !A^{\perp} \\
(\exists X.B)^{\perp} &= \forall X.B^{\perp} & (\forall X.B)^{\perp} &= \exists X.B^{\perp} \\
1^{\perp} &= \perp & \perp^{\perp} &= 1 \\
0^{\perp} &= \top & \top^{\perp} &= 0
\end{array}
$$

The dual of a propositional variable, $X^{\perp}$, is part of the syntax. Multiplicatives are dual to each other, as are additives, exponentials, and quantifiers.

Duality is an involution, $(A^{\perp})^{\perp} = A$.

Substitution. Write $B\{A/X\}$ to denote substitution of A for X in B. Substitution of a proposition for a dual propositional variable results in the dual of the proposition.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Assuming $X \neq Y$, define

$$X\{A/X\} = A \qquad X^{\perp}\{A/X\} = A^{\perp}$$
$$Y\{A/X\} = Y \qquad Y^{\perp}\{A/X\} = Y^{\perp}$$

The remaining clauses are standard, for instance $(A \otimes B)\{C/X\} = A\{C/X\} \otimes B\{C/X\}$.

Duality preserves substitution, $B\{A/X\}^{\perp} = B^{\perp}\{A/X\}$.

Environments. Let Γ, Δ, Θ range over environments associating names to propositions, where each name is distinct. Assuming $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, with $x_i \neq x_j$ whenever $i \neq j$, we write $\mathsf{fn}(\Gamma) = \{x_1, \ldots, x_n\}$ for the names in Γ, and $\mathsf{fv}(\Gamma) = \mathsf{fv}(A_1) \cup \cdots \cup \mathsf{fv}(A_n)$ for the free propositional variables in Γ. Order in environments is ignored. Environments use linear maintenance, so two environments may be combined only if they contain distinct names: writing Γ, Δ implies $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset$.

Processes. Our process calculus is a variant on the π-calculus (Milner *et al.* 1992). Processes are defined by the following grammar:

$P, Q, R ::=$	
$x \leftrightarrow y$	link
$\nu x{:}A.(P \mid Q)$	parallel composition
$x[y].(P \mid Q)$	output
$x(y).P$	input
$x[\mathsf{inl}].P$	left selection
$x[\mathsf{inr}].P$	right selection
$x.\mathsf{case}(P, Q)$	choice
$!x(y).P$	server accept
$?x[y].P$	client request
$x[A].P$	output a type
$x(X).P$	input a type
$x[].0$	empty output
$x().P$	empty input
$x.\mathsf{case}()$	empty choice

In $\nu x{:}A.(P \mid Q)$, name x is bound in P and Q, in $x[y].(P \mid Q)$, name y is bound in P (but not in Q), and in $x(y).P$, $?x[y].P$, and $!x(y).P$, name y is bound in P. We write $\mathsf{fn}(P)$ for the free names in process P. In $x(X).P$, propositional variable X is bound in P.

The form $x \leftrightarrow y$ denotes forwarding, where every message received on x is retransmitted on y, and every message received on y is retransmitted on x. Square brackets indicate output and round brackets indicate input; unlike π-calculus, both output and input names are bound. The forms $x(y).P$ and $!x(y).P$ in our calculus behave like the same forms in π-calculus, while both forms $x[y].P$ and $?x[y].P$ in our calculus behave like the form $\nu y.x\langle y\rangle.P$ in π-calculus.

Alternative notion. A referee suggested, in line with one tradition for π-calculus, choosing the notation $\overline{x}(y).P$ in place of $x[y].P$. We avoid this alternative because overlines can be hard to spot, while the distinction between round and square brackets is clear.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

$$\frac{}{w{\leftrightarrow}x \vdash w : A^{\perp}, x : A}\ \mathsf{Ax} \qquad \frac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^{\perp}}{\nu x{:}A.(P \mid Q) \vdash \Gamma, \Delta}\ \mathsf{Cut}$$

$$\frac{P \vdash \Gamma, y : A \quad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B}\ \otimes \qquad \frac{R \vdash \Theta, y : A, x : B}{x(y).R \vdash \Theta, x : A \parr B}\ \parr$$

$$\frac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B}\ \oplus_1 \qquad \frac{P \vdash \Gamma, x : B}{x[\mathsf{inr}].P \vdash \Gamma, x : A \oplus B}\ \oplus_2 \qquad \frac{Q \vdash \Delta, x : A \quad R \vdash \Delta, x : B}{x.\mathsf{case}(Q,R) \vdash \Delta, x : A \,\&\, B}\ \&$$

$$\frac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}\ ! \qquad \frac{Q \vdash \Delta, y : A}{?x[y].Q \vdash \Delta, x : ?A}\ ?$$

$$\frac{Q \vdash \Delta}{Q \vdash \Delta, x : ?A}\ \mathsf{Weaken} \qquad \frac{Q \vdash \Delta, x : ?A, x' : ?A}{Q\{x/x'\} \vdash \Delta, x : ?A}\ \mathsf{Contract}$$

$$\frac{P \vdash \Gamma, x : B\{A/X\}}{x[A].P \vdash \Gamma, x : \exists X.B}\ \exists \qquad \frac{Q \vdash \Delta, x : B}{x(X).Q \vdash \Delta, x : \forall X.B}\ \forall\ (X \notin \mathsf{fv}(\Delta))$$

$$\frac{}{x[].0 \vdash x : 1}\ 1 \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot}\ \bot \qquad \text{(no rule for 0)} \qquad \frac{}{x.\mathsf{case}() \vdash \Gamma, x : \top}\ \top$$

Fig. 1. CP, classical linear logic as a session-typed process calculus.

Judgements. The rules for assigning session types to processes are shown in Figure 1. Judgements take the form

$$P \vdash x_1 : A_1, \ldots, x_n : A_n$$

indicating that process P communicates along each channel named x_i obeying the protocol specified by A_i. Erasing the process and the channel names from the above yields

$$\vdash A_1, \ldots, A_n$$

and applying this erasure to the rules in Figure 1 yields the rules of classical linear logic, as given by Girard (1987).

3.1 Structural rules

The calculus has two structural rules, Axiom and Cut. We do not list Exchange explicitly, since order in environments is ignored.

The axiom is:

$$\frac{}{w{\leftrightarrow}x \vdash w : A^{\perp}, x : A}\ \mathsf{Ax}$$

We interpret the axiom as forwarding. A name input along w is forwarded as output along x, and *vice versa*, so types of the two channels must be dual. Bellin & Scott (1994) restrict the axiom to propositional variables, replacing A by X and $w{\leftrightarrow}x$ by the π-calculus term $w(y).x\langle y\rangle.0$. Whereas we forward any number of times and in either direction, they forward only once and from X to $X^{\perp}$.

The cut rule is:

$$\frac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^{\perp}}{\nu x{:}A.(P \mid Q) \vdash \Gamma, \Delta}\ \mathsf{Cut}$$

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

(Swap)

$$\frac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^{\perp}}{\nu x{:}A.(P \mid Q) \vdash \Gamma, \Delta}\ \mathsf{Cut} \quad \equiv \quad \frac{Q \vdash \Delta, x : A^{\perp} \quad P \vdash \Gamma, x : A}{\nu x{:}A^{\perp}.(Q \mid P) \vdash \Gamma, \Delta}\ \mathsf{Cut}$$

(Assoc)

$$\frac{\dfrac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^{\perp}, y : B}{\nu x.(P \mid Q) \vdash \Gamma, \Delta, y : B}\ \mathsf{Cut} \qquad R \vdash \Theta, y : B^{\perp}}{\nu y.(\nu x.(P \mid Q) \mid R) \vdash \Gamma, \Delta, \Theta}\ \mathsf{Cut} \quad \equiv$$

$$\frac{P \vdash \Gamma, x : A \qquad \dfrac{Q \vdash \Delta, x : A^{\perp}, y : B \quad R \vdash \Theta, y : B^{\perp}}{\nu y.(Q \mid R) \vdash \Delta, \Theta, x : A^{\perp}}\ \mathsf{Cut}}{\nu x.(P \mid \nu y.(Q \mid R)) \vdash \Gamma, \Delta, \Theta}\ \mathsf{Cut}$$

(AxCut)

$$\frac{\dfrac{}{w{\leftrightarrow}x \vdash w : A^{\perp}, x : A}\ \mathsf{Ax} \qquad P \vdash \Gamma, x : A^{\perp}}{\nu x.(w{\leftrightarrow}x \mid P) \vdash \Gamma, w : A^{\perp}}\ \mathsf{Cut} \quad \Longrightarrow \quad P\{w/x\} \vdash \Gamma, w : A^{\perp}$$

Fig. 2. Structural cut equivalences and reduction for CP.

Following Abramsky (1994) and Bellin & Scott (1994), we interpret Cut as a symmetric operation combining parallel composition with name restriction. Process P communicates along channel x obeying protocol A, while process Q communicates along the same channel x obeying the dual protocol $A^{\perp}$. Duality guarantees that sends and selections in P match with receives and choices in Q, and *vice versa*. Communications along Γ and Δ are disjoint, so P and Q are restricted to communicate with each other only along x. If communication could occur along two channels rather than one, this could lead to races or deadlock. (When we discuss exponentials, we will see that Γ and Δ may share channels of type $?B$, for some B. Such channels are only used to communicate with replicable servers, so it remains the case that the only communication between P and Q is along x.)

Observe that, despite writing $\nu x : A$ in the syntax, the type of x differs in P and Q – it is A in the former but $A^{\perp}$ in the latter. Including the type A in the syntax for Cut guarantees that given the type of each free name in the term, each term has a unique type derivation. To save ink and eyestrain, the type is omitted when it is clear from the context.

Cut elimination corresponds to process reduction. Figure 2 shows two equivalences on cuts, and one reduction that simplifies a cut against an axiom, each specified in terms of derivation trees; from which we read off directly the corresponding equivalence or reduction on processes. We write $\equiv$ for equivalences and $\Longrightarrow$ for reductions; both relations are reflexive and transitive. Equivalence (Swap) states that a cut is symmetric:

$$\nu x{:}A.(P \mid Q) \equiv \nu x{:}A^{\perp}.(Q \mid P)$$

It serves the same role as the π-calculus structural equivalence for symmetry, $P \mid Q \equiv Q \mid P$. Equivalence (Assoc) permits reordering cuts:

$$\nu y.(\nu x.(P \mid Q) \mid R) \equiv \nu x.(P \mid \nu y.(Q \mid R))$$

It serves the same role as the π-calculus structural equivalences for associativity, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, and scope extrusion, $(\nu x.P) \mid Q \equiv \nu x.(P \mid Q)$ when $x \notin P$.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Reduction (AxCut) simplifies a cut against an axiom.

$$\nu x.(w{\leftrightarrow}x \mid P) \Longrightarrow P\{w/x\}$$

We write $P\{w/x\}$ to denote substitution of w for x in P.

Alternative notation. The confusion of giving different types to x in P and Q might be avoided by picking a different syntax $\nu x{\leftrightarrow}y.(P \mid Q)$ with the typing rule

$$\frac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, y : A^{\perp}}{\nu x{\leftrightarrow}y.(P \mid Q) \vdash \Gamma, \Delta}\ \mathsf{Cut}$$

Here instead of using x as the distinguished free name of type A in P and type $A^{\perp}$ in Q, we retain x as the distinguished free name of type A in P but use y as the distinguished free name of type $A^{\perp}$ in Q. Thus, $\nu x{\leftrightarrow}y.(P \mid Q)$ in the alternative notation corresponds to $\nu x.(P \mid (Q\{x/y\}))$ in our notation. We avoid this alternative so as not to proliferate names.

3.2 Output and input

The multiplicative connectives $\otimes$ and $\parr$ are dual. We interpret $A \otimes B$ as the session type of a channel which outputs an A and then behaves as a B, and $A \parr B$ as the session type of a channel which inputs an A and then behaves as a B.

The rule for output is:

$$\frac{P \vdash \Gamma, y : A \quad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B}\ \otimes$$

Processes P and Q act on disjoint sets of channels. Process P communicates along channel y obeying protocol A, while process Q communicates along channel x obeying protocol B. The composite process $x[y].(P \mid Q)$ communicates along channel x obeying protocol $A \otimes B$; it allocates a fresh channel y, transmits y along x, and then concurrently executes P and Q. Disjointness of P and Q ensures that there is no further entangling between x and y, which guarantees freedom from races and deadlock.

The rule for input is:

$$\frac{R \vdash \Theta, y : A, x : B}{x(y).R \vdash \Theta, x : A \parr B}\ \parr$$

Process R communicates along channel y obeying protocol A and along channel x obeying protocol B. The composite process $x(y).R$ communicates along channel x obeying protocol $A \parr B$; it receives name y along x, and then executes R. Unlike with output, the single process R communicates with both x and y. It is safe to permit the same process to communicate with x and y on the input side because there is no further entangling of x with y on the output side, explaining the claim that disentangling x from y on output guarantees freedom from races and deadlock.

For output, channel x has type B in the component process Q but type $A \otimes B$ in the composite process $x[y].(P \mid Q)$. For input, channel x has type B in the component process R but type $A \parr B$ in the composite process $x(y).R$. One may regard the type of the channel evolving as communication proceeds, corresponding to the notion of the session type. Assigning the same channel name different types in the hypothesis and conclusion of a rule

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

is the telling twist added by Caires & Pfenning (2010) in contrast to the use of different variables in the hypothesis and conclusion followed by Abramsky (1994) and Bellin & Scott (1994).

The computational content of the logic is most clearly revealed in the principal cuts of each connective against its dual. Principal cut reductions are shown in Figure 3.

Cut of output $\otimes$ against input $\parr$ corresponds to communication, as shown in rule ($\beta_{\otimes\parr}$):

$$\nu x.(x[y].(P \mid Q) \mid x(y).R) \Longrightarrow \nu y.(P \mid \nu x.(Q \mid R))$$

In stating this rule, we take advantage of the fact that y is bound in both $x[y].P$ and $x(y).Q$ to assume that the same bound name y has been chosen in each; Pitts (2011) refers to this as the 'anti-Barendregt' convention.

Recall that $x[y].P$ in our notation corresponds to $\nu y.x\langle y\rangle.P$ in π-calculus. Thus, the above rule corresponds to the π-calculus reduction:

$$\nu x.(\nu y.x\langle y\rangle.(P \mid Q) \mid x(z).R) \Longrightarrow \nu y.P \mid \nu x.(Q \mid R\{z/y\})$$

This follows from from $x\langle y\rangle.P \mid x(z).R \Longrightarrow P \mid R\{z/y\}$, and the structural equivalences for scope extrusion, since $x \notin \mathsf{fn}(P)$.

One might wonder why the right-hand side of the above reduction is $\nu x.(P \mid \nu y.(Q \mid R))$ rather than $\nu y.(Q \mid \nu x.(P \mid R))$? The two are in fact equivalent by the use of the structural rules:

$$\begin{array}{lll} & \nu x{:}A.(P \mid \nu y{:}B.(Q \mid R)) & \\ \equiv & \nu x{:}A.(P \mid \nu y{:}B^{\perp}.(R \mid Q)) & (\mathsf{Swap}) \\ \equiv & \nu y{:}B^{\perp}.(\nu x{:}A.(P \mid R) \mid Q) & (\mathsf{Assoc}) \\ \equiv & \nu y{:}B.(Q \mid \nu x{:}A.(P \mid R)) & (\mathsf{Swap}) \end{array}$$

Hence, either term serves equally well as the right-hand side.

The apparent lack of symmetry between $A \otimes B$ and $B \otimes A$ may appear unsettling: the first means output A and then behave as B, the second means output B and then behave as A. The situation is similar to Cartesian product, where $B \times A$ and $A \times B$ differ but satisfy an isomorphism. Similarly, $A \otimes B$ and $B \otimes A$ are interconvertible.

$$\dfrac{\dfrac{\dfrac{}{w{\leftrightarrow}z \vdash w : B^{\perp}, z : B}\,\mathsf{Ax} \quad \dfrac{}{y{\leftrightarrow}x \vdash y : A^{\perp}, x : A}\,\mathsf{Ax}}{x[z].(w{\leftrightarrow}z \mid y{\leftrightarrow}x) \vdash w : B^{\perp}, y : A^{\perp}, x : B \otimes A}\,\otimes}{w(y).x[z].(w{\leftrightarrow}z \mid y{\leftrightarrow}x) \vdash w : A^{\perp} \parr B^{\perp}, x : B \otimes A}\,\parr$$

Let flip_{wx} be the term in the conclusion of the above derivation. Given an arbitrary derivation ending in $P \vdash \Gamma, w : A \otimes B$, one may replace $A \otimes B$ with $B \otimes A$ as follows:

$$\dfrac{P \vdash \Gamma, w : A \otimes B \quad \mathsf{flip}_{wx} \vdash w : A^{\perp} \parr B^{\perp}, x : B \otimes A}{\nu w.(P \mid \mathsf{flip}_{wx}) \vdash \Gamma, x : B \otimes A}\,\mathsf{Cut}$$

Here process P communicates along w obeying the protocol $A \otimes B$, outputting A and then behaving as B. Composing P with Q yields the process that communicates along x obeying the protocol $B \otimes A$, outputting B and then behaving as A.

The multiplicative units are 1 for $\otimes$ and $\perp$ for $\parr$. We interpret 1 as the session type of the channel that transmits an empty ouput, and $\perp$ as the session type of the channel that accepts an empty input. These are related by duality: $1^{\perp} = \perp$. Their rules are shown in Figure 1.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

$(\beta_{\otimes\parr})$

$$\dfrac{\dfrac{P \vdash \Gamma, y : A \quad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B}\otimes \quad \dfrac{R \vdash \Theta, y : A^{\perp}, x : B^{\perp}}{x(y).R \vdash \Theta, x : A^{\perp} \parr B^{\perp}}\parr}{\nu x.(x[y].(P \mid Q) \mid x(y).R) \vdash \Gamma, \Delta, \Theta}\,\mathsf{Cut} \quad \Longrightarrow$$

$$\dfrac{P \vdash \Gamma, y : A \qquad \dfrac{Q \vdash \Delta, x : B \quad R \vdash \Theta, y : A^{\perp}, x : B^{\perp}}{\nu x.(Q \mid R) \vdash \Delta, \Theta, y : A^{\perp}}\,\mathsf{Cut}}{\nu y.(P \mid \nu x.(Q \mid R)) \vdash \Gamma, \Delta, \Theta}\,\mathsf{Cut}$$

$(\beta_{\oplus\&})$

$$\dfrac{\dfrac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B}\oplus_1 \quad \dfrac{Q \vdash \Delta, x : A^{\perp} \quad R \vdash \Delta, x : B^{\perp}}{x.\mathsf{case}(Q,R) \vdash \Delta, x : A^{\perp} \,\&\, B^{\perp}}\,\&}{\nu x.(x[\mathsf{inl}].P \mid x.\mathsf{case}(Q,R)) \vdash \Gamma, \Delta}\,\mathsf{Cut} \quad \Longrightarrow$$

$$\dfrac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^{\perp}}{\nu x.(P \mid Q) \vdash \Gamma, \Delta}\,\mathsf{Cut}$$

$(\beta_{!?})$

$$\dfrac{\dfrac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}\,! \quad \dfrac{Q \vdash \Delta, y : A^{\perp}}{?x[y].Q \vdash \Delta, x : ?A^{\perp}}\,?}{\nu x.(!x(y).P \mid ?x[y].Q) \vdash ?\Gamma, \Delta}\,\mathsf{Cut} \quad \Longrightarrow \quad \dfrac{P \vdash ?\Gamma, y : A \quad Q \vdash \Delta, y : A^{\perp}}{\nu y.(P \mid Q) \vdash ?\Gamma, \Delta}\,\mathsf{Cut}$$

$(\beta_{!W})$

$$\dfrac{\dfrac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}\,! \quad \dfrac{Q \vdash \Delta}{Q \vdash \Delta, x : ?A^{\perp}}\,\mathsf{Weaken}}{\nu x.(!x(y).P \mid Q) \vdash ?\Gamma, \Delta}\,\mathsf{Cut} \quad \Longrightarrow \quad \dfrac{Q \vdash \Delta}{Q \vdash ?\Gamma, \Delta}\,\mathsf{Weaken}$$

$(\beta_{!C})$

$$\dfrac{\dfrac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}\,! \quad \dfrac{Q \vdash \Delta, x : ?A, x' : ?A}{Q\{x/x'\} \vdash \Delta, x : ?A}\,\mathsf{Contract}}{\nu x.(!x(y).P \mid Q\{x/x'\}) \vdash ?\Gamma, \Delta}\,\mathsf{Cut} \quad \Longrightarrow$$

$$\dfrac{\dfrac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}\,! \quad \dfrac{\dfrac{P' \vdash ?\Gamma', y' : A}{!x'(y').P' \vdash ?\Gamma', x' : !A}\,! \quad Q \vdash \Delta, x : ?A^{\perp}, x' : ?A^{\perp}}{\nu x'.(!x'(y').P' \mid Q) \vdash ?\Gamma', \Delta, x : ?A^{\perp}}\,\mathsf{Cut}}{\dfrac{\nu x.(!x(y).P \mid \nu x'.(!x'(y').P' \mid Q)) \vdash ?\Gamma, ?\Gamma', \Delta}{\nu x.(!x(y).P \mid \nu x'.(!x'(y).P \mid Q)) \vdash ?\Gamma, \Delta}\,\mathsf{Contract}}\,\mathsf{Cut}$$

$(\beta_{\exists\forall})$

$$\dfrac{\dfrac{P \vdash \Gamma, x : B\{A/X\}}{x[A].P \vdash \Gamma, x : \exists X.B}\,\exists \quad \dfrac{Q \vdash \Delta, x : B^{\perp}}{x(X).Q \vdash \Delta, x : \forall X.B^{\perp}}\,\forall}{\nu x.(x[A].P \mid x(X).Q) \vdash \Gamma, \Delta}\,\mathsf{Cut} \quad \Longrightarrow$$

$$\dfrac{P \vdash \Gamma, x : B\{A/X\} \quad Q\{A/X\} \vdash \Delta, x : B^{\perp}\{A/X\}}{\nu x.(P \mid Q\{A/X\}) \vdash \Gamma, \Delta}\,\mathsf{Cut}$$

$(\beta_{1\perp})$

$$\dfrac{\dfrac{}{x[].0 \vdash x : 1}\,1 \quad \dfrac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \perp}\,\perp}{\nu x.(x[].0 \mid x().P) \vdash \Gamma}\,\mathsf{Cut} \quad \Longrightarrow \quad P \vdash \Gamma$$

$(\beta_{0\top})$

(no rule for 0 with $\top$)

Fig. 3. Principal cut reductions for CP.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Cut of empty output 1 against empty input $\bot$ corresponds to an empty communication, as shown in rule $(\beta_{1\bot})$:

$$\nu x.(x[].0 \mid x().P) \Longrightarrow P$$

This rule resembles reduction of a nilary communication in the polyadic π-calculus.

Example. We give a series of examples inspired by Internet commerce, based on similar examples in Caires & Pfenning (2010). Our first example is that of a sale, where the client sends a product name and credit card number to a server, which returns a receipt. Define:

$$\begin{aligned}
\mathsf{Buy} &\stackrel{\text{def}}{=} \mathsf{Name} \otimes (\mathsf{Credit} \otimes (\mathsf{Receipt}^{\bot} \parr \bot)) \\
\mathsf{Sell} &\stackrel{\text{def}}{=} \mathsf{Name}^{\bot} \parr (\mathsf{Credit}^{\bot} \parr (\mathsf{Receipt} \otimes 1)) \\
\mathsf{buy}_x &\stackrel{\text{def}}{=} x[u].(\mathsf{put\text{-}name}_u \mid x[v].(\mathsf{put\text{-}credit}_v \mid x(w).x().\mathsf{get\text{-}receipt}_w)) \\
\mathsf{sell}_x &\stackrel{\text{def}}{=} x(u).x(v).x[w].(\mathsf{compute}_{u,v,w} \mid x[].0)
\end{aligned}$$

Here Name is the type of product names; Credit is the type of credit card numbers; Receipt is the type of receipts; $\mathsf{put\text{-}name}_u$ transmits on u the name of a product, say 'tea'; $\mathsf{put\text{-}credit}_v$ transmits on v a credit card number; $\mathsf{compute}_{u,v,w}$ accepts name u and credit card v, and computes a receipt, which it transmits on w; $\mathsf{get\text{-}receipt}_w$ accepts receipt w, and continues with the buyer's business; Γ specifies other channels used by the client; and Θ specifies other channels used by the server.

Observe that $\mathsf{Buy} = \mathsf{Sell}^{\bot}$ and

$$\frac{\mathsf{buy}_x \vdash \Gamma, x : \mathsf{Buy} \quad \mathsf{sell}_x \vdash \Theta, x : \mathsf{Sell}}{\nu x.(\mathsf{buy}_x \mid \mathsf{sell}_x) \vdash \Gamma, \Theta}\ \mathsf{Cut}$$

By three applications of $(\beta_{\otimes\parr})$ and one of $(\beta_{1\bot})$, we have

$$\begin{aligned}
&\nu x.(\mathsf{buy}_x \mid \mathsf{sell}_x) \Longrightarrow \\
&\qquad \nu u.(\mathsf{put\text{-}name}_u \mid \nu v.(\mathsf{put\text{-}credit}_v \mid \nu w.(\mathsf{compute}_{u,v,w} \mid \mathsf{get\text{-}receipt}_w)))
\end{aligned}$$

illustrating the interaction of output and input.

Example. As a further example, illustrating use of the units 1 and $\bot$, we consider a way to express two parallel computations. We introduce a primitive computation

$$\mathsf{par}_{y,z} \vdash y : 1, z : 1$$

that sends a signal along both y and z in parallel. Then we can derive a term that executes two processes $P \vdash \Gamma$ and $Q \vdash \Delta$ in parallel, as follows:

$$\dfrac{\dfrac{\mathsf{par}_{y,z} \vdash y : 1, z : 1 \quad \dfrac{P \vdash \Gamma}{y().P \vdash y : \bot, \Gamma}\ \bot}{\nu y.(\mathsf{par}_{y,z} \mid y().P) \vdash z : 1, \Gamma}\ \mathsf{Cut} \quad \dfrac{Q \vdash \Delta}{z().Q \vdash z : \bot, \Delta}\ \bot}{\nu z.(\nu y.(\mathsf{par}_{y,z} \mid y().P) \mid z().Q) \vdash \Gamma, \Delta}\ \mathsf{Cut}$$

In what follows, we abbreviate the above derivation as

$$\frac{P \vdash \Gamma \quad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ \mathsf{Mix}$$

We will see that Mix has a logical interpretation in Section 6.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

3.3 Selection and choice

The additive connectives $\oplus$ and $\&$ are dual. We interpret $A \oplus B$ as the session type of a channel which selects from either an A or a B, and $A \,\&\, B$ as the session type of a channel which offers a choice of either an A or a B.

The rule for left selection is:

$$\frac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B}\ \oplus_1$$

Process P communicates along channel x obeying protocol A. The composite process $x[\mathsf{inl}].P$ communicates along channel x obeying protocol $A \oplus B$; it transmits along x a request to select the left option from a choice, and then executes process P. The rule for right selection is symmetric.

The rule for choice is:

$$\frac{Q \vdash \Delta, x : A \quad R \vdash \Delta, x : B}{x.\mathsf{case}(Q,R) \vdash \Delta, x : A \,\&\, B}\ \&$$

The composite process $x.\mathsf{case}(Q,R)$ communicates along channel x obeying protocol $A \,\&\, B$; it receives a selection along channel x and executes either process Q or R accordingly.

For selection, channel x has type A in the component process P and type $A \oplus B$ in the composite process $x[\mathsf{inl}].P$. For choice, channel x has type A in the component process Q, type B in the component process R, and type $A \,\&\, B$ in the composite process $x.\mathsf{case}(Q,R)$. Again, one may regard the type of the channel evolving as communication proceeds, corresponding to the notion of session type.

Cut of selection $\oplus$ against choice $\&$ corresponds to picking an alternative as shown in rule $(\beta_{\oplus\&})$:

$$x[\mathsf{inl}].P \mid x.\mathsf{case}(Q,R) \Longrightarrow \nu x.(P \mid Q)$$

The rule to select the right option is symmetric.

The additive units are 0 for $\oplus$ and $\top$ for $\&$. We interpret 0 as the session type of a channel that selects from among no alternatives, and $\bot$ as the session type of a channel that offers a choice among no alternatives. These are related by duality: $0^{\bot} = \top$. Their rules are shown in Figure 1. There is no rule for 0 because it is impossible to select from no alternatives. Hence, there is also no reduction for a cut of an empty selection against an empty choice as shown in Figure 3.

Example. We extend our previous example to offer a choice of two operations, selling an item or quoting a price. To start with, we specify the second form of Internet commerce, quoting a price, where the client sends a product name to the server, which returns its price. Define:

$$\begin{aligned}
\mathsf{Shop} &\stackrel{\text{def}}{=} (\mathsf{Name} \otimes (\mathsf{Price}^{\bot} \parr \bot)) \\
\mathsf{Quote} &\stackrel{\text{def}}{=} (\mathsf{Name}^{\bot} \parr (\mathsf{Price} \otimes 1)) \\
\mathsf{shop}_x &\stackrel{\text{def}}{=} x[u].(\mathsf{put\text{-}name}_u \mid x(v).\mathsf{get\text{-}price}_v)) \\
\mathsf{quote}_x &\stackrel{\text{def}}{=} x(u).x[v].(\mathsf{lookup}_{u,v} \mid x[].0)
\end{aligned}$$

Here Name is the type of product names; Price is the type of prices; $\mathsf{put\text{-}name}_u$ transmits on u the name of a product, say 'tea'; $\mathsf{lookup}_{u,v}$ accepts name u and looks up the price,

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

which it transmits along v; get-price accepts price v, and continues with the requester's business; Δ specifies other channels used by the client; and Θ specifies other channels used by the server. Apart from the distinguished channel x, sell_X and quote_X use the *same* other channels, while buy_X and shop_x may use *different* other channels.

Observe that $\mathsf{Shop} = \mathsf{Quote}^\perp$ and

$$\frac{\mathsf{shop}_x \vdash \Delta, x : \mathsf{Shop} \quad \mathsf{quote}_x \vdash \Theta, x : \mathsf{Quote}}{\nu x.(\mathsf{shop}_x \mid \mathsf{quote}_x) \vdash \Delta, \Theta}\ \mathsf{Cut}$$

By two applications of $(\beta_{\otimes\parr})$ and one of $(\beta_{1\perp})$, we have

$$\nu x.(\mathsf{shop}_x \mid \mathsf{quote}_x) \Longrightarrow \nu u.(\mathsf{put\text{-}name}_u \mid \nu v.(\mathsf{lookup}_{u,v} \mid \mathsf{get\text{-}price}_v))$$

further illustrating the interaction of output and input.

We now combine the two servers into one that offers a choice of either service, and promote each of the previous clients into one that first selects the appropriate service and then behaves as before. Define:

$$\begin{array}{lcl} \mathsf{Select} & \stackrel{\mathsf{def}}{=} & \mathsf{Buy} \oplus \mathsf{Shop} \\ \mathsf{Choice} & \stackrel{\mathsf{def}}{=} & \mathsf{Sell} \mathbin{\&} \mathsf{Quote} \\ \mathsf{select\text{-}buy}_x & \stackrel{\mathsf{def}}{=} & x[\mathsf{inl}].\mathsf{buy}_x \\ \mathsf{select\text{-}shop}_x & \stackrel{\mathsf{def}}{=} & x[\mathsf{inr}].\mathsf{shop}_x \\ \mathsf{choice}_x & \stackrel{\mathsf{def}}{=} & x.\mathsf{case}(\mathsf{sell}_x, \mathsf{quote}_x) \end{array}$$

Observe that $\mathsf{Select} = (\mathsf{Choice})^\perp$ and

$$\frac{\mathsf{select\text{-}buy}_x \vdash \Gamma, x : \mathsf{Choose} \quad \mathsf{choice}_x \vdash \Theta, x : \mathsf{Offer}}{\nu x.(\mathsf{select\text{-}buy}_x \mid \mathsf{choice}_x) \vdash \Gamma, \Theta}\ \mathsf{Cut}$$

By one application of $(\beta_{\oplus\&})$ we have

$$\nu x.(\mathsf{select\text{-}buy}_x \mid \mathsf{choice}_x) \Longrightarrow \nu x.(\mathsf{buy}_x \mid \mathsf{sell}_x)$$

illustrating the interaction of selection and choice. Similarly, the above judgement also holds if we replace $\mathsf{select\text{-}buy}_x$ with $\mathsf{select\text{-}shop}_x$ and Γ with Δ, and we have

$$\nu x.(\mathsf{select\text{-}shop}_x \mid \mathsf{choice}_x) \Longrightarrow \nu x.(\mathsf{shop}_x \mid \mathsf{quote}_x)$$

illustrating the other selection.

3.4 Servers and clients

The exponential connectives ! and ? are dual. We interpret $!A$ as the session type of a server that will repeatedly accept an A, and interpret $?A$ as the session type of a collection of clients that may each request an A. A server must be impartial, providing the same service to each client, whereas clients may pass different requests to the same server. Hence, $!A$ offers uniform behaviour, while $?A$ accumulates diverse behviours.

The rule for servers is:

$$\frac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}\ !$$

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Process P communicates along channel y obeying protocol A. The composite process $!x(y).P$ communicates along channel x obeying the protocol $!A$; it receives y along x, and then spawns a fresh copy of P to execute. All channels used by P other than y must obey a protocol of the form $?B$, for some B, to ensure that replicating P respects the type discipline. Intuitively, a process may only provide a replicable service if it is implemented by communicating only with other processes that provide replicable services.

There are three rules for clients corresponding to the fact that a server may have one, none, or many clients. The three rules correspond to the rules of classical linear logic for dereliction, weakening, and contraction.

The first rule is for a single client.

$$\frac{Q \vdash \Delta, y : A}{?x[y].Q \vdash \Delta, x : ?A}\ ?$$

Process Q communicates along channel y obeying protocol A. The composite process $?x[y].Q$ communicates along channel x obeying protocol $?A$; it allocates a fresh channel y, transmits y along x, and then executes process Q. Cut of rule ! against rule ? corresponds to spawning a single copy of a server to communicate with a client, as shown in rule $(\beta_{!?})$:

$$\nu x.(!x(y).P \mid ?x[y].Q) \Longrightarrow \nu y.(P \mid Q)$$

The second rule is for no clients.

$$\frac{Q \vdash \Delta}{Q \vdash \Delta, x : ?A}\ \mathsf{Weaken}$$

A process Q that does not communicate along any channel obeying protocol A may be regarded as communicating along a channel obeying protocol $?A$. Cut of rule ! against Weaken corresponds to garbage collection, deallocating a server that has no clients, as shown in rule $(\beta_{!W})$:

$$\nu x.(!x(y).P \mid Q) \Longrightarrow Q, \quad \text{if } x \notin \mathsf{fn}(Q)$$

The third rule aggregates multiple clients.

$$\frac{Q \vdash \Delta, x : ?A, x' : ?A}{Q\{x/x'\} \vdash \Delta, x : ?A}\ \mathsf{Contract}$$

Process Q communicates along two channels x and x' both obeying protocol $?A$. Process $Q\{x/x'\}$ is identical to Q, save all occurrences of x' have been renamed to x; it communicates along a single channel x obeying protocol $?A$. Cut of rule ! against Contract corresponds to replicating a server, as shown in rule $(\beta_{!C})$:

$$\nu x.(!x(y).P \mid Q\{x/x'\}) \Longrightarrow \nu x.(!x(y).P \mid \nu x'.(!x'(y).P \mid Q))$$

The type derivation on the right-hand side of rule $(\beta_{!C})$ applies Contract once for each free name in Γ. The derivation is written using the following priming convention: we assume that to each name z_i there is associated another name z_i', and we write P' for the process identical to P, save that each free name z_i in P has been replaced by z_i'; that is, if $\mathsf{fn}(P) = \{z_1, \ldots, z_n\}$ then $P' = P\{z_1'/z_1, \ldots, z_n'/z_n\}$.

Example. We further extend our example so that the server offers a replicated service to multiple clients. Presume that in the preceding examples the channels Θ used by sell_x,

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

quote_x, and choice_x are in fact of the form $?\Theta$ so that each process is implemented by communicating only with other processes that provide replicable services. Define:

$$\begin{aligned}
\mathsf{Client} &\stackrel{\text{def}}{=} ?\mathsf{Select} \\
\mathsf{Server} &\stackrel{\text{def}}{=} !\mathsf{Choice} \\
\mathsf{client}_x &\stackrel{\text{def}}{=} ?x(y).\mathsf{select\text{-}buy}_y \mid ?x(y).\mathsf{select\text{-}shop}_y \\
\mathsf{server}_x &\stackrel{\text{def}}{=} !x(y).\mathsf{choice}_y
\end{aligned}$$

Here the combination of the two processes in client_x is formed using the Mix rule, as discussed in the second example of Section 3.2.

Observe that $\mathsf{Client} = \mathsf{Server}^{\perp}$ and

$$\frac{\mathsf{client}_x \vdash \Gamma, \Delta, x : \mathsf{Client} \quad \mathsf{server}_x \vdash ?\Theta, x : \mathsf{Server}}{\nu x.(\mathsf{client}_x \mid \mathsf{server}_x) \vdash \Gamma, \Delta, ?\Theta}\ \mathsf{Cut}$$

By one application of $(\beta_{!C})$ and two of $(\beta_{!?})$ we have

$$\nu x.(\mathsf{client}_y \mid \mathsf{server}_y) \Longrightarrow (\nu y.\mathsf{select\text{-}buy}_y \mid \mathsf{choice}_y) \mid (\nu y'.\mathsf{select\text{-}shop}_{y'} \mid \mathsf{choice}_{y'})$$

illustrating the interaction of a replicable server with multiple clients. Note that pushing the two instances of choice inside the instance of Mix requires two uses of the structural rule Assoc from Section 3.1 and two uses of the commuting conversion κ_0 discussed in Section 3.6.

Alternative notation. A referee notes that weakening and contraction could be given explicit notation rather than implicit, for instance, using $?x[].Q$ to denote weakening and $?x[x',x''].Q$ to denote contraction, yielding type rules

$$\frac{Q \vdash \Delta}{?x[].Q \vdash \Delta, x : ?A}\ \mathsf{Weaken}$$

and

$$\frac{Q \vdash \Delta, x' : ?A, x'' : ?A}{?x[x',x''].Q \vdash \Delta, x : ?A}\ \mathsf{Contract}$$

while reduction rules $(\beta_{!W})$ and $(\beta_{!C})$ become

$$\nu x.(!x(y).P \mid ?x[].Q) \Longrightarrow ?z_1[].\cdots.?z_n[].Q$$

and

$$\begin{aligned}
&\nu x.(!x(y).P \mid ?x[x',x''].Q) \Longrightarrow \\
&\qquad ?z_1[z_1',z_1''].\cdots.?z_n[z_n',z_n''].\nu x'.(!x'(y').P' \mid \nu x''.(!x''(y'').P'' \mid Q))
\end{aligned}$$

where $\mathsf{fn}(P) = \{y, z_1, \ldots, z_n\}$. We avoid this alternative because the implicit notation is more convenient.

3.5 Polymorphism

The quantifiers $\exists$ and $\forall$ are dual. We interpret $\exists X.B$ as the session type of a channel that instantiates propositional variable X to a given proposition, and interpret $\forall X.B$ as the session type of a process that generalises over X. These correspond to type application

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

and type abstraction in polymorphic λ-calculus, or to sending and receiving types in the polymorphic π-calculus of Turner (1995).

The rule for instantiation is:

$$\frac{P \vdash \Gamma, x : B\{A/X\}}{x[A].P \vdash \Gamma, x : \exists X.B}\ \exists$$

Process P communicates along channel x obeying protocol $B\{A/X\}$. The composite process $x[A].P$ communicates along channel x obeying protocol $\exists X.B$; it transmits a representation of A along x, and then executes P.

The rule for generalisation is:

$$\frac{Q \vdash \Delta, x : B}{x(X).Q \vdash \Delta, x : \forall X.B}\ \forall\ \ (X \notin \mathsf{fv}(\Delta))$$

Process Q communicates along channel x obeying protocol B. The composite process $x(X).Q$ communicates along channel x obeying protocol $\forall X.B$; it receives a description of a proposition along channel x, binds the proposition to the propositional variable X, and then executes Q.

Cut of instantiation $\exists$ against generalisation $\forall$ corresponds to transmitting a representation of a proposition, as shown in rule ($\beta_{\exists\forall}$):

$$\nu x.(x[A].P \mid x(X).Q) \Longrightarrow \nu x.(P \mid Q\{A/X\})$$

This rule behaves similarly to beta reduction of a type abstraction against a type application in polymorphic λ-calculus, or communication of a type in the polymorphic π-calculus.

Example. Quantification supports a definition of the Church numerals in our system. Define

$$\begin{aligned}
\mathsf{Church} &\stackrel{\text{def}}{=} \forall X.?(X \otimes X^{\perp}) \parr (X^{\perp} \parr X)\\
\mathsf{zero}_x &\stackrel{\text{def}}{=} x(X).x(s).x(z).z{\leftrightarrow}x\\
\mathsf{one}_x &\stackrel{\text{def}}{=} x(X).x(s).x(z).?s[f].f[a].(a{\leftrightarrow}z \mid f{\leftrightarrow}x)\\
\mathsf{two}_x &\stackrel{\text{def}}{=} x(X).x(s).x(z).?s[f].f[a].(a{\leftrightarrow}z \mid ?s[g].g[b].(f{\leftrightarrow}b \mid g{\leftrightarrow}x))
\end{aligned}$$

Observe that if we define $A \multimap B = A^{\perp} \parr B$ then the type of the Church numerals may be rewritten as $\forall X.!(X \multimap X) \multimap (X \multimap X)$, which may appear more familiar. The terms zero_x, one_x, and two_x accept a type variable X, a process $s : ?(X \otimes X^{\perp})$ and a value $z : X^{\perp}$, and invoke s zero, one, or two times on z to return a value of type X. We may invoke the Church numerals by instantiating X, s, and z appropriately.

Define process

$$\mathsf{count}_{x,y} \vdash x : \mathsf{Church}^{\perp}, y : \mathsf{Nat}$$

that accepts a Church numeral on x and transmits the corresponding natural on y as follows:

$$\mathsf{count}_{x,y} \stackrel{\text{def}}{=} x[\mathsf{Nat}].x[s].(!s(f).f(a).\mathsf{incr}_{a,f} \mid x[z].(\mathsf{nought}_z \mid x{\leftrightarrow}y))$$

Here Nat is the type of natural numbers; process $\mathsf{incr}_{a,b} \vdash a : \mathsf{Nat}^{\perp}, b : \mathsf{Nat}$ accepts a natural along a and transmits a value one greater along b; and process $\mathsf{nought}_a \vdash a : \mathsf{Nat}$ transmits

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

the value zero along a. Then

$$\begin{aligned} \nu x.(\mathsf{zero}_x \mid \mathsf{count}_{x,y}) &\Longrightarrow \mathsf{nought}_y \\ \nu x.(\mathsf{one}_x \mid \mathsf{count}_{x,y}) &\Longrightarrow \nu z.(\mathsf{nought}_z \mid \mathsf{incr}_{z,y}) \\ \nu x.(\mathsf{two}_x \mid \mathsf{count}_{x,y}) &\Longrightarrow \nu a.(\nu z.(\mathsf{nought}_z \mid \mathsf{incr}_{z,a}) \mid \mathsf{incr}_{a,y}) \end{aligned}$$

These three processes transmit 0, 1, or 2, respectively, along y.

Similarly, define process

$$\mathsf{ping}_{x,y,w} \vdash x : \mathsf{Church}^{\perp}, y : ?\perp, w : 1$$

that accepts a Church numeral on x and transmits a corresponding number of signals along y, and when done transmits a signal along w as follows:

$$\mathsf{ping}_{x,y,w} \stackrel{\text{def}}{=} x[1].x[s].(!s[f].f(a).a().?y[u].u().f[].0 \mid x[z].(z[].0 \mid x().w[].0))$$

Then

$$\begin{aligned} \nu x.(\mathsf{zero}_x \mid \mathsf{ping}_{x,y,w}) &\Longrightarrow w[].0 \\ \nu x.(\mathsf{one}_x \mid \mathsf{ping}_{x,y,w}) &\Longrightarrow ?y[u].u().w[].0 \\ \nu x.(\mathsf{two}_x \mid \mathsf{ping}_{x,y,w}) &\Longrightarrow ?y[u].u().?y[v].v().w[].0 \end{aligned}$$

These three processes transmit 0, 1, or 2 signals, respectively, along y, and then a signal along w.

3.6 Commuting conversions

Commuting conversions are shown in Figures 4 and 5.

Each commuting conversion pushes a cut inside a communication operation. There are two conversions for $\otimes$, depending upon whether the cut pushes into the left or right branch. Each of the remaining logical operators has one conversion, with the exception of $\oplus$, which has two (only the left rule is shown, the right rule is symmetric); and the exception of 1 and 0, which have none.

An important aspect of CP is revealed by considering rule $(\kappa_{\otimes})$, which pushes cut inside input:

$$\nu z.(x(y).P \mid Q) \Longrightarrow x(y).\nu z.(P \mid Q)$$

On the left-hand side process Q may interact with the environment, while on the right-hand side Q is guarded by the input and cannot interact with the environment. In our setting, this is not problematic. If x is bound by an outer cut, then the guarding input is guaranteed to match a corresponding output at some point. If x is not bound by an outer cut, then we consider the process halted while it awaits external communication along x; compare this with the use of labelled transitions in Lemma 5.7 of Caires & Pfenning (2010).

3.7 Cut elimination

In addition to the rules of Figures 2–5, we add a standard rule relating reductions to structural equivalences:

$$\frac{P \equiv Q \quad Q \Longrightarrow R \quad R \equiv S}{P \Longrightarrow S}$$

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

$$(\kappa_{\otimes 1})\quad \frac{\dfrac{P \vdash \Gamma, y : A, z : C \quad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B, z : C}\otimes \quad R \vdash \Theta, z : C^{\perp}}{\nu z.(x[y].(P \mid Q) \mid R) \vdash \Gamma, \Delta, \Theta, x : A \otimes B}\,\mathsf{Cut} \quad \Longrightarrow$$

$$\frac{\dfrac{P \vdash \Gamma, y : A, z : C \quad R \vdash \Theta, z : C^{\perp}}{\nu z.(P \mid R) \vdash \Gamma, \Theta, y : A}\,\mathsf{Cut} \quad Q \vdash \Delta, x : B}{x[y].(\nu z.(P \mid R) \mid Q) \vdash \Gamma, \Delta, \Theta, x : A \otimes B}\otimes$$

$$(\kappa_{\otimes 2})\quad \frac{\dfrac{P \vdash \Gamma, y : A \quad Q \vdash \Delta, x : B, z : C}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B, z : C}\otimes \quad R \vdash \Theta, z : C^{\perp}}{\nu z.(x[y].(P \mid Q) \mid R) \vdash \Gamma, \Delta, \Theta, x : A \otimes B}\,\mathsf{Cut} \quad \Longrightarrow$$

$$\frac{P \vdash \Gamma, y : A \qquad \dfrac{Q \vdash \Delta, x : B, z : C \quad R \vdash \Theta, z : C^{\perp}}{\nu z.(Q \mid R) \vdash \Delta, \Theta, x : B}\,\mathsf{Cut}}{x[y].(P \mid \nu z.(Q \mid R)) \vdash \Gamma, \Delta, \Theta, x : A \otimes B}\otimes\,\mathsf{Cut}$$

$$(\kappa_{\parr})\quad \frac{\dfrac{P \vdash \Gamma, y : A, x : B, z : C}{x(y).P \vdash \Gamma, x : A \parr B, z : C}\parr \quad Q \vdash \Delta, z : C^{\perp}}{\nu z.(x(y).P \mid Q) \vdash \Gamma, \Delta, x : A \parr B}\,\mathsf{Cut} \quad \Longrightarrow$$

$$\frac{\dfrac{P \vdash \Gamma, y : A, x : B, z : C \quad Q \vdash \Delta, z : C^{\perp}}{\nu z.(P \mid Q) \vdash \Gamma, \Delta, y : A, x : B}\,\mathsf{Cut}}{x(y).\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : A \parr B}\parr\,\mathsf{Cut}$$

$$(\kappa_{\&})\quad \frac{\dfrac{P \vdash \Gamma, x : A, z : C \quad Q \vdash \Gamma, x : B, z : C}{x.\mathsf{case}(P,Q) \vdash \Gamma, x : A \,\&\, B, z : C}\,\& \quad R \vdash \Delta, z : C^{\perp}}{\nu z.(x.\mathsf{case}(P,Q) \mid R) \vdash \Gamma, \Delta, x : A \,\&\, B}\,\mathsf{Cut} \quad \Longrightarrow$$

$$\frac{\dfrac{P \vdash \Gamma, x : A, z : C \quad R \vdash \Delta, z : C^{\perp}}{\nu z.(P \mid R) \vdash \Gamma, \Delta, x : A}\,\mathsf{Cut} \quad \dfrac{Q \vdash \Gamma, x : B, z : C \quad R \vdash \Delta, z : C^{\perp}}{\nu z.(Q \mid R) \vdash \Gamma, \Delta, x : B}\,\mathsf{Cut}}{x.\mathsf{case}(\nu z.(P \mid R), \nu z.(Q \mid R)) \vdash \Gamma, \Delta, x : A \,\&\, B}\,\&\,\mathsf{Cut}$$

$$(\kappa_{\oplus})\quad \frac{\dfrac{P \vdash \Gamma, x : A, z : C}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B, z : C}\oplus \quad Q \vdash \Delta, z : C^{\perp}}{\nu z.(x[\mathsf{inl}].P \mid Q) \vdash \Gamma, \Delta, x : A \oplus B}\,\mathsf{Cut} \quad \Longrightarrow \quad \frac{\dfrac{P \vdash \Gamma, x : A, z : C \quad Q \vdash \Delta, z : C^{\perp}}{\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : A}\,\mathsf{Cut}}{x[\mathsf{inl}].\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : A \oplus B}\oplus$$

Fig. 4. Commuting conversions for CP, Part I.

And we add congruence rules for cuts:

$$\frac{P_1 \Longrightarrow P_2}{\nu x.(P_1 \mid Q) \Longrightarrow \nu x.(P_2 \mid Q)} \qquad \frac{Q_1 \Longrightarrow Q_2}{\nu x.(P \mid Q_1) \Longrightarrow \nu x.(P \mid Q_2)}$$

These rules are standard in treatments of cut elimination. We do not add congruences for other operators; see below.

CP satisfies subject reduction: well-typed processes reduce to well-typed processes.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

$$(\kappa_!)\quad \frac{\dfrac{P \vdash ?\Gamma, y : A, z : ?C}{!x(y).P \vdash ?\Gamma, x : !A, z : ?C}\,! \qquad Q \vdash ?\Delta, z : !C^\perp}{\nu z.(!x(y).P \mid Q) \vdash ?\Gamma, ?\Delta, x : !A}\,\mathsf{Cut} \quad\Longrightarrow\quad \frac{\dfrac{P \vdash ?\Gamma, y : A, z : ?C \qquad Q \vdash ?\Delta, z : !C^\perp}{\nu z.(P \mid Q) \vdash ?\Gamma, ?\Delta, y : A}\,\mathsf{Cut}}{!x(y).\nu z.(P \mid Q) \vdash ?\Gamma, ?\Delta, x : !A}\,!$$

$$(\kappa_?)\quad \frac{\dfrac{P \vdash \Gamma, y : A, z : C}{?x(y).P \vdash \Gamma, x : ?A, z : C}\,? \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(?x(y).P \mid Q) \vdash \Gamma, \Delta, x : ?A}\,\mathsf{Cut} \quad\Longrightarrow\quad \frac{\dfrac{P \vdash \Gamma, y : A, z : C \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(P \mid Q) \vdash \Gamma, \Delta, y : A}\,\mathsf{Cut}}{?x(y).\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : ?A}\,?$$

$$(\kappa_\exists)\quad \frac{\dfrac{P \vdash \Gamma, x : B\{A/X\}, z : C}{x[A].P \vdash \Gamma, x : \exists X.B, z : C}\,\exists \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(x[A].P \mid Q) \vdash \Gamma, \Delta, x : \exists X.B}\,\mathsf{Cut} \quad\Longrightarrow\quad \frac{\dfrac{P \vdash \Gamma, x : B\{A/X\}, z : C \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : B\{A/X\}}\,\mathsf{Cut}}{x[A].\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : \exists X.B}\,\exists$$

$$(\kappa_\forall)\quad \frac{\dfrac{P \vdash \Gamma, x : B, z : C}{x(X).P \vdash \Gamma, x : \forall X.B, z : C}\,\forall \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(x(X).P \mid Q) \vdash \Gamma, \Delta, x : \forall X.B}\,\mathsf{Cut} \quad\Longrightarrow\quad \frac{\dfrac{P \vdash \Gamma, x : B, z : C \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : B}\,\mathsf{Cut}}{x(X).\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : \forall X.B}\,\forall$$

$$(\kappa_\perp)\quad \frac{\dfrac{P \vdash \Gamma, z : C}{x().P \vdash \Gamma, x : \perp, z : C}\,\perp \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(x().P \mid Q) \vdash \Gamma, \Delta, x : \perp}\,\mathsf{Cut} \quad\Longrightarrow\quad \frac{\dfrac{P \vdash \Gamma, z : C \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(P \mid Q) \vdash \Gamma, \Delta}\,\mathsf{Cut}}{x().\nu z.(P \mid Q) \vdash \Gamma, \Delta, x : \perp}\,\perp$$

$$(\kappa_\top)\quad \frac{\dfrac{}{x.\mathsf{case}() \vdash \Gamma, x : \top, z : C}\,\top \qquad Q \vdash \Delta, z : C^\perp}{\nu z.(x.\mathsf{case}() \mid Q) \vdash \Gamma, \Delta, x : \perp}\,\mathsf{Cut} \quad\Longrightarrow\quad \frac{}{x.\mathsf{case}() \vdash \Gamma, \Delta, x : \top}\,\top$$

Fig. 5. Commutative conversions for CP, Part II.

Theorem 1
If $P \vdash \Gamma$ and $P \Longrightarrow Q$ then $Q \vdash \Gamma$.

Proof sketch: Figures 2–5 contain the relevant proofs. □

Say process P is a *cut* if it has the form $\nu x.(Q \mid R)$ for some x, Q, and R. CP satisfies top-level cut elimination: every process reduces to a process that is not a cut.

Theorem 2
If $P \vdash \Gamma$ then there exists a Q such that $P \Longrightarrow Q$ and Q is not a cut.

Proof sketch: Each rule is either Ax, Cut, or a logical rule. If P is a cut, there are three possibilities: If one side of the cut uses the axiom, apply AxCut. If one side of the cut

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

is itself a cut, recursively eliminate the cut. In the remaining cases, either both sides are logical rules that act on the cut variable, in which case a principal reduction of Figure 3 applies, or at least one side is a logical rule acting on a variable other than the cut variable, in which case a commuting reduction of Figure 4 or 5 applies. Since we support impredicative polymorphism, where a polymorphic type may be instantiated by a polymorphic type, care is required in formulating the induction to ensure termination, but this is standard (Gallier 1990). □

This result resembles the Principal Lemma of Cut Elimination (Girard *et al.* 1989, Section 13.2), which eliminates a final cut rule, possibly replacing it with (smaller) cuts further up the proof tree. Top-level cut elimination corresponds to lack of deadlock; it ensures that any process can reduce until it needs to perform an external communication.

If our goal was to eliminate all cuts, we would need to introduce additional congruence rules, such as

$$\frac{P \Longrightarrow Q}{x(y).P \Longrightarrow x(y).Q}$$

and similarly for each operator. Such rules do not correspond well to our notion of computation on processes, so we omit them; this is analogous to the usual practice of not permitting reduction under lambda.

4 A session-typed functional language

This section presents GV, a session-typed functional language based on one devised by Gay & Vasconcelos (2010), and presents its translation into CP.

Our presentation of GV differs in some particulars from that of Gay & Vasconcelos (2010). Most notably, our system is guaranteed free from deadlock whereas theirs is not. Achieving this property requires some modifications to their system. We split their session type 'end' into two dual types '$\mathsf{end}_!$' and '$\mathsf{end}_?$', and we replace their constructs 'accept', 'request', and 'fork', by two new constructs 'with-connect-to' and 'terminate'.

A number of features of Gay & Vasconcelos (2010) are not echoed here. Their system is based on asynchronous buffered communication, they show that the size required of asynchronous buffers can be bounded by analysing session types, and they support recursive functions, recursive session types, and subtyping. We omit these contributions for simplicity, but see no immediate difficulty in extending our results to include them. Of course, adding recursive terms or recursive session types may remove the property that all programs terminate.

For simplicity, we also omit a number of other possible features. We do not consider base types, which are straightforward. We also do not consider how to add replicated servers with multiple clients along the lines suggested by ! and ? in CP, or how to add polymorphism along the lines suggested by ∃ and ∀ in CP, but both extensions appear straightforward.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Session types. Session types are defined by the following grammar:

$$
\begin{array}{ll}
S ::= & \\
\quad !T.S & \text{output value of type } T \text{ then behave as } S \\
\quad ?T.S & \text{input value of type } T \text{ then behave as } S \\
\quad \oplus\{l_i : S_i\}_{i\in I} & \text{select from behaviours } S_i \text{ with label } l_i \\
\quad \&\{l_i : S_i\}_{i\in I} & \text{offer choice of behaviours } S_i \text{ with label } l_i \\
\quad \mathsf{end}_! & \text{terminator, convenient for use with output} \\
\quad \mathsf{end}_? & \text{terminator, convenient for use with input}
\end{array}
$$

Let S range over session types, and let T, U, V range over types. Session type $!T.S$ describes a channel along which a value of type T may be sent and which subsequently behaves as S. Dually, $?T.S$ describes a channel along which a value of type T may be received and which subsequently behaves as S. Session type $\oplus\{l_i : S_i\}_{1\in I}$ describes a channel along which one of the distinct labels l_i may be sent and which subsequently behaves as S_i. Dually, $\&\{l_i : S_i\}_{1\in I}$ describes a channel along which one of the labels l_i may be received, and which subsequently behaves as S_i. Finally, $\mathsf{end}_!$ and $\mathsf{end}_?$ describe channels that cannot be used for further communication. As we will see, it is convenient to use one if the last action on the channel is a send, and the other if the last action on the channel is a receive.

Types. Types are defined by the following grammar:

$$
\begin{array}{ll}
T, U, V ::= & \\
\quad S & \text{session type (linear)} \\
\quad T \otimes U & \text{tensor product (linear)} \\
\quad T \multimap U & \text{function (linear)} \\
\quad T \to U & \text{function (unlimited)} \\
\quad \mathsf{Unit} & \text{unit (unlimited)}
\end{array}
$$

Every session type is also a type, but not conversely. Types are formed from session types, tensor product, two forms of function space, and a unit for tensor product.

Each type is classified as linear or unlimited:

$$\mathsf{lin}(S) \quad \mathsf{lin}(T \otimes U) \quad \mathsf{lin}(T \multimap U) \quad \mathsf{un}(T \to U) \quad \mathsf{un}(\mathsf{Unit})$$

Here $\mathsf{lin}(T)$ denotes a type that is linear, and $\mathsf{un}(T)$ denotes a type that is unlimited. Session types, tensor, and one type of function are limited; the other type of function and unit are unlimited. Unlimited types support weakening and contraction, while linear types do not. Unlimited types correspond to those written with ! in CP.

Duals. Each session type S has a dual $\overline{S}$, defined as follows:

$$
\begin{aligned}
\overline{!T.S} &= ?T.\overline{S} \\
\overline{?T.S.} &= !T.\overline{S} \\
\overline{\oplus(l_i : S_i)_{i\in I}} &= \&(l_i : \overline{S}_i)_{i\in I} \\
\overline{\&(l_i : S_i)_{i\in I}} &= \oplus(l_i : \overline{S}_i)_{i\in I} \\
\overline{\mathsf{end}_!} &= \mathsf{end}_? \\
\overline{\mathsf{end}_?} &= \mathsf{end}_!
\end{aligned}
$$

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

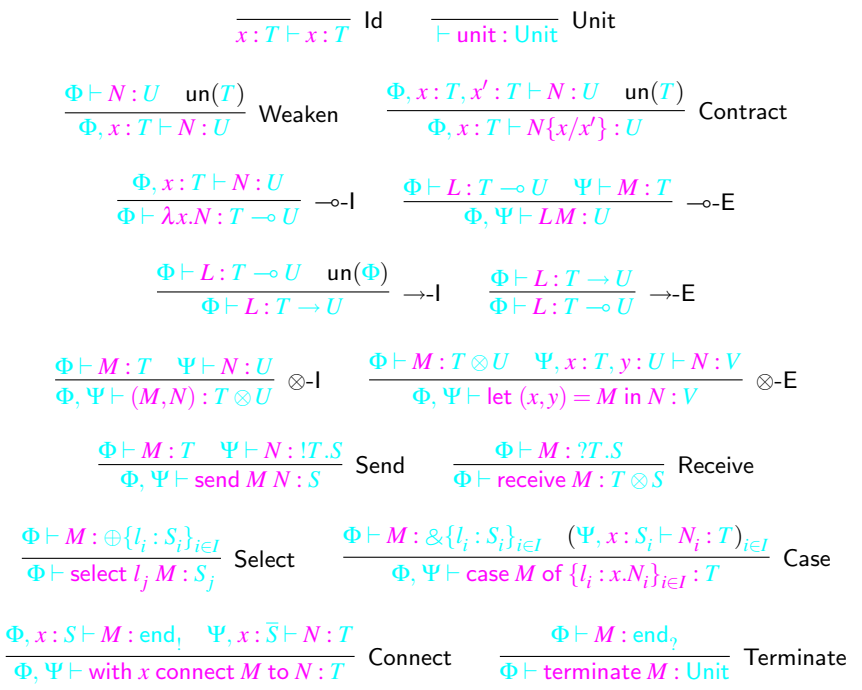

Fig. 6. GV, a session-typed functional language.

Input is dual to output, selection is dual to choice, and the two terminators are dual. Duality between input and output does not take the dual of the type.

Duality is an involution, $\overline{\overline{S}} = S$.

Environments. We let Φ, Ψ range over environments associating variables to types. Write $\mathsf{un}(\Phi)$ to indicate that each type in Φ is unlimited. As in Section 3, order in environments is ignored and we use linear maintenance.

Terms. Terms are defined by the following grammar:

$L, M, N ::=$	
x	identifier
unit	unit constant
$\lambda x.N$	function abstraction
LM	function application
(M,N)	pair construction
$\mathsf{let}\ (x,y) = M\ \mathsf{in}\ N$	pair deconstruction
$\mathsf{send}\ M\ N$	send value M on channel N
$\mathsf{receive}\ M$	receive from channel M
$\mathsf{select}\ l\ M$	select label l on channel M
$\mathsf{case}\ M\ \mathsf{of}\ \{l_i : x.N_i\}_{i \in I}$	offer choice on channel M
$\mathsf{with}\ x\ \mathsf{connect}\ M\ \mathsf{to}\ N$	connect M to N by channel x
$\mathsf{terminate}\ M$	terminate input

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

The first six operations specify a linear λ-calculus, and the remaining six specify communication along a channel.

The terms are best understood in conjunction with their type rules, as shown in Figure 6. The rules for variables, unit, weakening, contraction, function abstraction and application, and pair construction and deconstruction are standard. Functions are either limited or unlimited. As usual, function abstraction may produce an unlimited function only if all of its free variables are of unlimited type. Following Gay & Vasconcelos (2010), we do not give a separate rule for application of an unlimited function, but instead give a rule permitting an unlimited function to be treated as a linear function, which may then be applied using the rule for linear function application.

For simplicity, we do not require that each term has a unique type. In particular, a λ-expression where all free variables have unlimited type may be given either linear or unlimited function type. In a practical system, one might introduce subtyping and arrange that each term has a unique smallest type.

The rule for output is

$$\frac{\Phi \vdash M : T \quad \Psi \vdash N : !T.S}{\Phi, \Psi \vdash \mathsf{send}\ M\ N : S}\ \mathsf{Send}$$

Channels are managed linearly, so each operation on channels takes the channel before the operation as an argument, and returns the channel after the operation as the result. Executing 'send M N' outputs the value M of type T along channel N of session type $!T.S$, and returns the updated channel, which after the output has session type S.

The rule for input is

$$\frac{\Phi \vdash M : ?T.S}{\Phi \vdash \mathsf{receive}\ M : T \otimes S}\ \mathsf{Receive}$$

Executing 'receive M' inputs a value from channel M of session type $?T.S$, and returns a pair consisting of the input value of type T, and the updated channel, which after the input has session type S. The returned pair must be linear because it contains a session type, which is linear.

Gay & Vasconcelos (2010) treat 'send' and 'receive' as function constants, and require two versions of 'send' to cope with complications arising from currying. We treat 'send' and 'receive' as language constructs, which avoids the need for two versions of 'send'. Thanks to the rules for limited and unlimited function abstraction, $\lambda x.\lambda y.\mathsf{send}\ x\ y$ has type $T \multimap !T.S \multimap S$ and also type $T \to !T.S \multimap S$ when $\mathsf{un}(T)$.

Select and Case are similar to Send and Receive, and standard.

The rule to create new channels is:

$$\frac{\Phi, x : S \vdash M : \mathsf{end}_! \quad \Psi, x : \overline{S} \vdash N : T}{\Phi, \Psi \vdash \mathsf{with}\ x\ \mathsf{connect}\ M\ \mathsf{to}\ N : T}\ \mathsf{Connect}$$

Executing 'with x connect M to N' creates a new channel x with session type S, where x is used at type S within term M and at the dual type $\overline{S}$ within term N. The two terms M and N are evaluated concurrently. As is usual when forking off a value, only one of the two sub-terms returns a value that is passed to the rest of the program. The left sub-term returns the exhausted channel, which has type $\mathsf{end}_!$. The right sub-term returns a value of type T that is passed on to the rest of the program.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Finally, we require a rule to terminate the other channel:

$$\frac{\Phi \vdash M : \mathsf{end}_?}{\Phi \vdash \mathsf{terminate}\ M : \mathsf{Unit}}\ \mathsf{Terminate}$$

Executing 'terminate M' evaluates term M which returns an exhausted channel of type $\mathsf{end}_?$ (of linear type), which is deallocated. The expression returns the value of type Unit, which is an unlimited type and hence may be discarded.

The constructs for Connect and Terminate between them deallocate two ends of a channel. The system is designed, so it is convenient to use $\mathsf{end}_!$ on a channel whose last operation is Send or Select, and $\mathsf{end}_?$ on a channel whose last operation is Receive or Case.

Usually, session-typed systems make end an unlimited type that is self-dual, but the formulation here fits better with Classical Linear Logic (CLL). A variation where end is a linear type requiring explicit deallocation is considered by Vasconcelos (2011).

One might consider alternative designs, say to replace Connect by an operation that creates a channel and returns both ends of it in a pair of type $S \otimes \overline{S}$, or to replace Terminate by an operation that takes a pair of type $\mathsf{end}_! \otimes \mathsf{end}_?$ and returns unit. However, both of these designs are difficult to translate into CP, which suggests they may suffer from deadlock.

Example. We reprise the example of a sale from Section 3.2, where the client sends a product name and credit card number to a server, which returns a receipt. Define:

$$\begin{array}{lcl}
\mathsf{Buy} & \stackrel{\text{def}}{=} & !\mathsf{Name}.!\mathsf{Credit}.?\mathsf{Receipt}.\mathsf{end}_? \\
\mathsf{Sell} & \stackrel{\text{def}}{=} & ?\mathsf{Name}.?\mathsf{Credit}.!\mathsf{Receipt}.\mathsf{end}_! \\
\mathsf{buy}_x & \stackrel{\text{def}}{=} & \mathsf{let}\ u = \mathsf{get\text{-}name}\ \mathsf{in} \\
 & & \mathsf{let}\ x_1 = \mathsf{send}\ u\ x\ \mathsf{in} \\
 & & \mathsf{let}\ v = \mathsf{get\text{-}credit}\ \mathsf{in} \\
 & & \mathsf{let}\ x_2 = \mathsf{send}\ v\ x_1\ \mathsf{in} \\
 & & \mathsf{let}\ (w, x_3) = \mathsf{receive}\ x_2\ \mathsf{in} \\
 & & \mathsf{let}\ \mathsf{unit} = \mathsf{terminate}\ x_3\ \mathsf{in} \\
 & & \mathsf{put\text{-}receipt}\ w \\
\mathsf{sell}_x & \stackrel{\text{def}}{=} & \mathsf{let}\ (u, x_1) = \mathsf{receive}\ x\ \mathsf{in} \\
 & & \mathsf{let}\ (v, x_2) = \mathsf{receive}\ x_1\ \mathsf{in} \\
 & & \mathsf{let}\ w = \mathsf{compute}\ u\ v\ \mathsf{in} \\
 & & \mathsf{send}\ w\ x_2
\end{array}$$

Here $\mathsf{let}\ (x, y) = M\ \mathsf{in}\ N$ is pair deconstruction, as introduced above; and $\mathsf{let}\ x = M\ \mathsf{in}\ N$ stands for $(\lambda x.N)\, M$; and $\mathsf{let}\ \mathsf{unit} = M\ \mathsf{in}\ N$ stands for $(\lambda x.N)\, M$ where $x : \mathsf{Unit}$ does not apear in N. Let Name be the type of product names; Credit be the type of credit card numbers; Receipt be the type of receipts; and Rest be the type of the rest of the computation. The first three types are unlimited, and Rest may be either linear or unlimited. Assume environments Φ and Ψ define variables of the given types:

$$\begin{array}{lcl}
\Phi & \stackrel{\text{def}}{=} & \mathsf{get\text{-}name} : \mathsf{Name},\ \mathsf{get\text{-}credit} : \mathsf{Credit},\ \mathsf{put\text{-}receipt} : \mathsf{Receipt} \to \mathsf{Rest} \\
\Psi & \stackrel{\text{def}}{=} & \mathsf{compute} : \mathsf{Name} \to \mathsf{Credit} \to \mathsf{Receipt}
\end{array}$$

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Observe that $\mathsf{Buy} = \overline{\mathsf{Sell}}$ and

$$\frac{\Psi, x : \mathsf{Sell} \vdash \mathsf{sell}_x : \mathsf{end}_! \quad \Phi, x : \mathsf{Buy} \vdash \mathsf{buy}_x : \mathsf{Rest}}{\Psi, \Phi \vdash \mathsf{with}\ x\ \mathsf{connect}\ \mathsf{sell}_x\ \mathsf{to}\ \mathsf{buy}_x : \mathsf{Rest}}\ \mathsf{Connect}$$

4.1 Translation

The translation of GV into CP is given in Figures 7 and 8.

Session types. The translation of session types is as follows:

$$\begin{aligned}
[\![!T.S]\!] &= [\![T]\!]^\perp \parr [\![S]\!] \\
[\![?T.S]\!] &= [\![T]\!] \otimes [\![S]\!] \\
[\![\oplus\{l_i : S_i\}_{i\in I}]\!] &= [\![S_1]\!] \mathbin{\&} \cdots \mathbin{\&} [\![S_n]\!], \quad I = \{1,\ldots,n\} \\
[\![\&\{l_i : S_i\}_{i\in I}]\!] &= [\![S_1]\!] \oplus \cdots \oplus [\![S_n]\!], \quad I = \{1,\ldots,n\} \\
[\![\mathsf{end}_!]\!] &= \bot \\
[\![\mathsf{end}_?]\!] &= 1
\end{aligned}$$

This translation is surprising in that each operator translates to the dual of what one might expect! The session type for output in GV, $!T.S$ is translated into $\parr$, the connective that is interpreted as input in CP, and the session type for input in GV, $?T.S$ is translated into $\otimes$, the connective that is interpreted as output in CP. Similarly, $\oplus$ and $\&$ in GV translate, respectively, to $\&$ and $\oplus$ in CP. Finally, $\mathsf{end}_!$ and $\mathsf{end}_?$ in GV translate, respectively, to $\bot$ and 1 in CP, the units for $\parr$ and $\otimes$.

The intuitive explanation of this duality is that Send and Receive in GV take channels as *arguments* whereas the interpretation of the connectives in CP is for channels as *results*. Indeed, the send operation takes a value and a channel, and sends the value to that channel – in other words, the channel must *input* the value. Dually, the receive operation takes a channel and returns a value – in other words, the channel must *output* the value. A similar inversion occurs with respect to Select and Case.

Recall that duality on session types in GV leaves the types of sent and received values unchanged:

$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S.} = !T.\overline{S}$$

Conversely, the translation of these operations takes the dual of the sent value, but not the received value:

$$[\![!T.S]\!] = [\![T]\!]^\perp \parr [\![S]\!] \qquad [\![?T.S]\!] = [\![T]\!] \otimes [\![S]\!]$$

In classical linear logic, $A \multimap B = A^\perp \parr B$ so the right-hand side of the first line could alternatively be written $[\![T]\!] \multimap [\![S]\!]$. Accordingly, and as one would hope, the translation preserves duality: $[\![\overline{S}]\!] = [\![S]\!]^\perp$.

Types. The translation of types is as follows:

$$\begin{aligned}
[\![T \multimap U]\!] &= [\![T]\!]^\perp \parr [\![U]\!] \\
[\![T \to U]\!] &= !([\![T]\!]^\perp \parr [\![U]\!]) \\
[\![T \otimes U]\!] &= [\![T]\!] \otimes [\![U]\!] \\
[\![\mathsf{Unit}]\!] &= !\top
\end{aligned}$$

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

$$\left[\!\left[\frac{}{x : T \vdash x : T}\ \mathsf{Id}\right]\!\right] z \quad = \quad \frac{}{x{\leftrightarrow}z \vdash x : [\![T]\!]^{\perp}, z : [\![T]\!]}\ \mathsf{Ax}$$

$$\left[\!\left[\frac{}{\vdash \mathsf{unit} : \mathsf{Unit}}\ \mathsf{Unit}\right]\!\right] z \quad = \quad \frac{\dfrac{}{y.\mathsf{case}() \vdash y : \top}\ \top}{!z(y).y.\mathsf{case}() \vdash z : !\top}\ !$$

$$\left[\!\left[\frac{\Phi \vdash N : U \quad \mathsf{un}(T)}{\Phi, x : T \vdash N : U}\ \mathsf{Weaken}\right]\!\right] z \quad = \quad \frac{[\![N]\!]z \vdash [\![\Phi]\!]^{\perp}, z : [\![U]\!]}{[\![N]\!]z \vdash [\![\Phi]\!]^{\perp}, x : [\![T]\!]^{\perp}, z : [\![U]\!]}\ \mathsf{Weaken}$$

$$\left[\!\left[\frac{\Phi, x : T, x' : T \vdash N : U \quad \mathsf{un}(T)}{\Phi, x : T \vdash N\{x/x'\} : U}\ \mathsf{Contract}\right]\!\right] z \quad = \quad \frac{[\![N]\!]z \vdash [\![\Phi]\!]^{\perp}, x : [\![T]\!]^{\perp}, x' : [\![T]\!]^{\perp}, z : [\![U]\!]}{[\![N\{x/x'\}]\!]z \vdash [\![\Phi]\!]^{\perp}, x : [\![T]\!]^{\perp}, z : [\![U]\!]}\ \mathsf{Contract}$$

$$\left[\!\left[\frac{\Phi, x : T \vdash N : U}{\Phi \vdash \lambda x.N : T \multimap U}\ \multimap\text{-}\mathsf{I}\right]\!\right] z \quad = \quad \frac{[\![N]\!]z \vdash [\![\Phi]\!]^{\perp}, x : [\![T]\!]^{\perp}, z : [\![U]\!]}{z(x).[\![N]\!]z \vdash [\![\Phi]\!]^{\perp}, z : [\![T]\!]^{\perp} \parr [\![U]\!]}\ \parr$$

$$\left[\!\left[\frac{\Phi \vdash L : T \multimap U \quad \Psi \vdash M : T}{\Phi, \Psi \vdash LM : U}\ \multimap\text{-}\mathsf{E}\right]\!\right] z \quad =$$

$$\frac{[\![L]\!]y \vdash [\![\Phi]\!]^{\perp}, y : [\![T]\!]^{\perp} \parr [\![U]\!] \quad \dfrac{[\![M]\!]x \vdash [\![\Psi]\!]^{\perp}, x : [\![T]\!] \quad \dfrac{}{y{\leftrightarrow}z \vdash y : [\![U]\!]^{\perp}, z : [\![U]\!]}\ \mathsf{Ax}}{y[x].([\![M]\!]x \mid y{\leftrightarrow}z) \vdash [\![\Psi]\!]^{\perp}, y : [\![T]\!] \otimes [\![U]\!]^{\perp}, z : [\![U]\!]}\ \otimes}{\nu y.([\![L]\!]y \mid y[x].([\![M]\!]x \mid y{\leftrightarrow}z)) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![U]\!]}\ \mathsf{Cut}$$

$$\left[\!\left[\frac{\Phi \vdash L : T \multimap U \quad \mathsf{un}(\Phi)}{\Phi \vdash L : T \to U}\ \to\text{-}\mathsf{I}\right]\!\right] z \quad = \quad \frac{[\![L]\!]y \vdash [\![\Phi]\!]^{\perp}, y : [\![T \multimap U]\!]}{!z(y).[\![L]\!]y \vdash [\![\Phi]\!]^{\perp}, z : ![\![T \multimap U]\!]}\ !$$

$$\left[\!\left[\frac{\Phi \vdash L : T \to U}{\Phi \vdash L : T \multimap U}\ \to\text{-}\mathsf{E}\right]\!\right] z \quad =$$

$$\frac{[\![L]\!]y \vdash [\![\Phi]\!]^{\perp}, y : ![\![T \multimap U]\!]^{\perp} \quad \dfrac{\dfrac{}{x{\leftrightarrow}z \vdash x : [\![T \multimap U]\!]^{\perp}, z : [\![T \multimap U]\!]}\ \mathsf{Ax}}{?y[x].x{\leftrightarrow}z \vdash y : ?[\![T \multimap U]\!]^{\perp}, z : [\![T \multimap U]\!]}\ ?}{\nu y.([\![L]\!]y \mid ?y[x].x{\leftrightarrow}z) \vdash [\![\Phi]\!]^{\perp}, z : [\![T \multimap U]\!]}\ \mathsf{Cut}$$

$$\left[\!\left[\frac{\Phi \vdash M : T \quad \Psi \vdash N : U}{\Phi, \Psi \vdash (M,N) : T \otimes U}\ \otimes\text{-}\mathsf{I}\right]\!\right] z \quad = \quad \frac{[\![M]\!]y \vdash [\![\Phi]\!]^{\perp}, y : [\![T]\!] \quad [\![N]\!]z \vdash [\![\Psi]\!]^{\perp}, z : [\![U]\!]}{z[y].([\![M]\!]y \mid [\![N]\!]z) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![T]\!] \otimes [\![U]\!]}\ \otimes$$

$$\left[\!\left[\frac{\Phi \vdash M : T \otimes U \quad \Psi, x : T, y : U \vdash N : V}{\Phi, \Psi \vdash \mathsf{let}\ (x,y) = M\ \mathsf{in}\ N : V}\ \otimes\text{-}\mathsf{E}\right]\!\right] z \quad =$$

$$\frac{[\![M]\!]y \vdash [\![\Phi]\!]^{\perp}, y : [\![T]\!] \otimes [\![U]\!] \quad \dfrac{[\![N]\!]z \vdash [\![\Psi]\!]^{\perp}, x : [\![T]\!]^{\perp}, y : [\![U]\!]^{\perp}, z : [\![V]\!]}{y(x).[\![N]\!]z \vdash [\![\Psi]\!]^{\perp}, y : [\![T]\!]^{\perp} \parr [\![U]\!]^{\perp}, z : [\![V]\!]}\ \parr}{\nu y.([\![M]\!]y \mid y(x).[\![N]\!]z) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![V]\!]}\ \mathsf{Cut}$$

Fig. 7. Translation from GV into CP, Part I.

Session types are also types, they are translated as above.

The right-hand side of the first equation could alternatively be written $[\![T]\!] \multimap [\![U]\!]$, showing that linear functions translate as standard.

The right-hand side of the second equation could alternatively be written $!([\![T]\!] \multimap [\![U]\!])$. There are two standard translations of intuitionistic logic into classical linear logic or,

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

$$\left[\!\!\left[\dfrac{\Phi \vdash M : T \quad \Psi \vdash N : !T.S}{\Phi, \Psi \vdash \mathsf{send}\ M\ N : S}\ \mathsf{Send}\right]\!\!\right] z \quad = \quad \dfrac{\dfrac{[\![M]\!] y \vdash [\![\Phi]\!]^{\perp}, y : [\![T]\!] \qquad \dfrac{}{x \leftrightarrow z \vdash x : [\![S]\!]^{\perp}, z : [\![S]\!]}\ \mathsf{Ax}}{x[y].([\![M]\!] y \mid x \leftrightarrow z) \vdash [\![\Phi]\!]^{\perp}, x : [\![T]\!] \otimes [\![S]\!]^{\perp}, z : [\![S]\!]}\ \otimes \qquad [\![N]\!] x \vdash [\![\Psi]\!]^{\perp}, x : [\![T]\!]^{\perp} \parr [\![S]\!]}{\nu x.(x[y].([\![M]\!] y \mid x \leftrightarrow z) \mid [\![N]\!] x) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![S]\!]}\ \mathsf{Cut}$$

$$\left[\!\!\left[\dfrac{\Phi \vdash M : ?T.S}{\Phi \vdash \mathsf{receive}\ M : T \otimes S}\ \mathsf{Receive}\right]\!\!\right] z \quad = \quad [\![M]\!] z \vdash [\![\Phi]\!]^{\perp}, z : [\![T]\!] \otimes [\![S]\!]$$

$$\left[\!\!\left[\dfrac{\Phi \vdash M : \oplus\{l_i : S_i\}_{i \in I}}{\Phi \vdash \mathsf{select}\ l_j\ M : S_j}\ \mathsf{Select}\right]\!\!\right] z \quad = \quad \dfrac{[\![M]\!] x \vdash [\![\Phi]\!]^{\perp}, x : [\![S_1]\!] \,\&\, \cdots \,\&\, [\![S_n]\!] \qquad \dfrac{\dfrac{}{x \leftrightarrow z \vdash x : [\![S_j]\!]^{\perp}, z : [\![S_j]\!]}\ \mathsf{Ax}}{x[\mathsf{in}_j].x \leftrightarrow z \vdash x : [\![S_1]\!]^{\perp} \oplus \cdots \oplus [\![S_n]\!]^{\perp}, z : [\![S_j]\!]}\ \oplus_{\mathsf{i}}}{\nu x.([\![M]\!] x \mid x[\mathsf{in}_j].x \leftrightarrow z) \vdash [\![\Phi]\!]^{\perp}, z : [\![S_j]\!]}\ \mathsf{Cut}$$

$$\left[\!\!\left[\dfrac{\Phi \vdash M : \&\{l_i : S_i\}_{i \in I} \quad (\Psi, x : S_i \vdash N_i : T)_{i \in I}}{\Phi, \Psi \vdash \mathsf{case}\ M\ \mathsf{of}\ \{l_i : x.N_i\}_{i \in I} : T}\ \mathsf{Case}\right]\!\!\right] z \quad =$$

$$\dfrac{[\![M]\!] x \vdash [\![\Phi]\!]^{\perp}, x : [\![S_1]\!] \oplus \cdots \oplus [\![S_n]\!] \qquad \dfrac{([\![N_i]\!] z \vdash [\![\Psi]\!]^{\perp}, x : [\![S_i]\!]^{\perp}, z : [\![T]\!])_{i \in I}}{x.\mathsf{case}([\![N_1]\!], \ldots, [\![N_n]\!]) \vdash x : [\![S_1]\!] \,\&\, \cdots \,\&\, [\![S_n]\!], z : [\![T]\!]}\ \&}{\nu x.([\![M]\!] x \mid x.\mathsf{case}([\![N_1]\!], \ldots, [\![N_n]\!])) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![T]\!]}\ \mathsf{Cut}$$

$$\left[\!\!\left[\dfrac{\Phi, x : S \vdash M : \mathsf{end}_! \quad \Psi, x : \overline{S} \vdash N : T}{\Phi, \Psi \vdash \mathsf{with}\ x\ \mathsf{connect}\ M\ \mathsf{to}\ N : T}\ \mathsf{Connect}\right]\!\!\right] z \quad =$$

$$\dfrac{\dfrac{[\![M]\!] y \vdash [\![\Phi]\!]^{\perp}, x : [\![S]\!]^{\perp}, y : \bot \qquad \dfrac{}{y[].0 \vdash y : 1}\ 1}{\nu y.([\![M]\!] y \mid y[].0) \vdash [\![\Phi]\!]^{\perp}, x : [\![S]\!]^{\perp}}\ \mathsf{Cut} \qquad [\![N]\!] z \vdash [\![\Psi]\!]^{\perp}, x : [\![S]\!], z : [\![T]\!]}{\nu x.(\nu y.([\![M]\!] y \mid y[].0) \mid [\![N]\!] z) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![T]\!]}\ \mathsf{Cut}$$

$$\left[\!\!\left[\dfrac{\Phi \vdash M : \mathsf{end}_?}{\Phi \vdash \mathsf{terminate}\ M : \mathsf{Unit}}\ \mathsf{Terminate}\right]\!\!\right] z \quad =$$

$$\dfrac{[\![M]\!] x \vdash [\![\Phi]\!]^{\perp}, x : 1 \qquad \dfrac{\dfrac{\dfrac{}{y.\mathsf{case}() \vdash y : \top}\ \top}{!z(y).y.\mathsf{case}() \vdash z : !\top}\ !}{x().!z(y).y.\mathsf{case}() \vdash x : 0, z : !\top}\ 0}{\nu x.([\![M]\!] x \mid x().!z(y).y.\mathsf{case}()) \vdash [\![\Phi]\!]^{\perp}, z : !\top}\ \mathsf{Cut}$$

Fig. 8. Translation from GV into CP, Part II.

equivalently, of λ-calculus into linear λ-calculus. Girard's (1987) original takes $(A \to B)^{\circ} = !A^{\circ} \multimap B^{\circ}$ and corresponds to call-by-name, while a lesser known alternative takes $(A \to B)^{*} = !(A^{*} \multimap B^{*})$ and corresponds to call-by-value (see Benton & Wadler, 1996 and Toninho *et al.*, 2012). The second is used here.

In classical linear logic, there is a bi-implication between 1 and $!\top$ (in many models, this bi-implication is an isomorphism), so the right-hand side of the last equation could alternatively be written 1, the unit for $\otimes$.

An unlimited type in GV translates to a type constructed with ! in CP: If $\mathsf{un}(T)$ then $[\![T]\!] = !A$, for some A.

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

Terms. Translation of terms is written in a continuation-passing style standard for translations of λ-calculi into process calculi. The translation of term M of type T is written $[\![M]\!]z$, where z is a channel of type $[\![T]\!]$; the process that translates M transmits the answer it computes along z. More precisely, if $\Phi \vdash M : T$ then $[\![M]\!]z \vdash [\![\Phi]\!]^{\perp}, z : [\![T]\!]$, where the Φ to the left of the turnstile in GV translates, as one might expect, to the dual $[\![\Phi]\!]^{\perp}$ on the right of the turn-style in CP.

The translation of terms is shown in Figures 7 and 8. Rather than simply giving a translation from terms of GV to terms of CP, we show the translation as taking type derivation trees to type derivation trees. Giving the translation on type derivation trees rather than terms has two advantages. Firstly, it eliminates any ambiguity arising from the fact, noted previously, that terms in GV do not have unique types. Secondly, it makes it easy to validate that the translation preserves types.

Figure 7 shows the translations for operations of a linear λ-calculus. A variable translates to an axiom, weakening and contraction translate to weakening and contraction. Both function abstraction and product deconstruction translate to input, and both function application and product construction translate to output. The translation of each elimination rule ($\multimap$-E, $\rightarrow$-E, and $\otimes$-E) also requires a use of Cut.

Figure 8 shows the translation for operations for communication. For purposes of the translation, it is convenient to work with n-fold analogues of $\oplus$ and $\&$, writing $\in_i$ for selection and $\mathsf{case}(P_1, \ldots, P_n)$ for choice.

Despite the inversion noted earlier in the translation of session types, the translation of Send involves an output operation of the form $x[y].(P \mid Q)$, the translation of Select involves a select operation of the form $x[\mathrm{in}_j].P$, the translation of Case involves a choice operation of the form $\mathsf{case}(Q_1, \ldots, Q_n)$, the translation of $\mathsf{end}_!$ in Connect involves an empty output of the form $y[\,].0$, and the translation of Terminate involves an empty input of the form $x().P$. Each of these translations also introduces a Cut, corresponding to communication with supplied channel. The translation of Receive is entirely trivial, but the corresponding input operation of the form $x(y).R$ appears in the translation of $\otimes$-E, which deconstructs the returned pair. Finally, the translation of Connect involves a Cut, which corresponds to introducing a channel for communication between the two sub-terms.

The translation preserves types.

Theorem 3
If $\Phi \vdash M : T$ then $[\![M]\!]x \vdash [\![\Phi]\!]^{\perp}, x : [\![T]\!]$.

Proof sketch: See Figures 7 and 8. □

We also claim that the translation preserves the intended semantics. The formal semantics of Gay & Vasconcelos (2010) is based on asynchronous buffered communication, which adds additional complications, so we leave a formal proof of correspondence between the two for the future work.

5 Related work

Session types. Session types were introduced by Honda (1993), and further extended by Takeuchi *et al.* (1994), Honda *et al.* (1998), and Yoshida & Vasconcelos (2007). Subtyping for session types is considered by Gay & Hole (2005), and the linear functional language

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

for session types considered in this paper was introduced by Gay & Vasconcelos (2010). Session types have been applied to describe operating system services by Fähndrich *et al.* (2006).

Deadlock freedom. Variations on session types that guarantee deadlock freedom are presented in Sumii & Kobayashi (1998) and Carbone & Debois (2010). Unlike CP, where freedom from deadlock follows from the relation to cut elimination, in the first it is ensured by introducing a separate partial order on time tags, and in the second by introducing a constraint on underlying dependency graphs.

Linear types for process calculus. A variety of linear type systems for process calculus are surveyed by Kobayashi (2002). Most of these systems look rather different than session types, but Kobayashi *et al.* (1996) present an embedding of session types into a variant of π-calculus with linear types for channels.

Linear proof search. Functional programming can be taken as arising from the Curry–Howard correspondence by associating program evaluation with proof normalisation. Analogously, logic programming can be taken as arising by associating program evaluation with proof search. Logic programming approaches based on linear logic give rise to systems with some similarities to CP (see Miller, 1992 and Kobayashi & Yonezawa, 1993, 1994, 1995).

Polymorphism. CP's support of polymorphism is based on the polymorphic π-calculus introduced by Turner (1995) and further discussed by Pierce & Turner (2000) and Pierce & Sangiorgi (2000). More recently, Caires *et al.* (2013) extended session types to polymorphism and established logical relations for parametricity. All of the above use explict polymorphism (Church-style). In contrast, Berger *et al.* (2005) introduced a polymorphically typed session calculus that uses implicit polymorphism (Curry-style).

Linear logic as a process calculus. Various interpretations of linear logic as a process calculus are proposed by Abramsky (1993, 1994) and Abramsky *et al.* (1996), the second of these being elaborated in detail by Bellin & Scott (1994).

This paper is inspired by a series of papers by Caires, Pfenning, Toninho, and Pérez. Caires and Pfenning (2010) firstly observed the correspondence relating formulas of linear logic to session types; its journal version is Caires *et al.* (2012b). Pfenning *et al.* (2011) extended the correspondence to dependent types in a stratified system, with concurrent communication at the outer level and a dependently typed functional language at the inner level. Pfenning *et al.* (2011) extended that system to support proof-carrying code and proof irrelevance. Toninho *et al.* (2012) explore encodings of λ-calculus into πDILL. Pérez *et al.* (2012) introduce logical relations on linear-typed processes to prove termination and contextual equivalences. Caires *et al.* (2012a) is the text of an invited talk at the Workshop on Types in Language Design and Implementation (TLDI), summarising much of the above.

Two additional papers have appeared since the International Conference on Functional Programming (ICFP) version of this paper. Caires *et al.* (2013) add polymorphism and

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

parametricity. Toninho *et al.* (2013) exploit monads to integrate a functional language with a session-typed process calculus.

Mazurak & Zdancewic (2010) present Lolliproc, which also offers a Curry–Howard interpretation of session types by relating the call/cc control operators to communication using a double-negation operator on types.

DILL versus CLL. Caires *et al.* (2012b) consider a variant of πDILL based on one-sided sequents of classical linear logic, which they call πCLL. Their πCLL is similar to CP, but differs in important particulars: its bookkeeping is more elaborate, using two zones: one linear and another intuitionitic; it has no axiom, so cannot easily support polymorphism; and it does not support reductions corresponding to the commuting conversions.

Caires *et al.* (2012b) state, they prefer a formulation based on DILL to one based on CLL, because DILL satisfies a locality property for replicated input, while CLL does not. Locality requires that names received along a channel may be used to send output but not to receive input, and is useful both from an implementation point of view and because a process calculus so restricted satisfies additional observational equivalences as shown by Merro & Sangiorgi (2004). Caires *et al.* (2012b) only restrict *replicated* input because restricting *all* input is too severe for a session-typed calculus. However, the good properties of locality have been studied only in the case where *all* input is prohibited on received names. It remains to be seen as to what extent the fact that DILL imposes locality for replicated names is significant.

In addition, in a private communication, Pfenning relayed that he believes DILL may be amenable to extension to dependent types, while he suspects CLL is not because strong sums become degenerate in some classical settings as shown by Herbelin (2005). However, linear logic is more amenable to constructive treatment than traditional classical logic as argued by Girard (1991), so it remains unclear to what extent CP, or πCLL, may support dependent types.

6 Conclusions

One reason that λ-calculus provides such a successful foundation for functional programming is that it includes both fragments that guarantee termination (typed λ-calculi) and fragments that can model any recursive function (untyped λ-calculus, or typed λ-calculi augmented with a general fix point operator). Indeed, the former can be seen as giving rise to the latter by considering recursive types with recursion in negative positions; untyped λ-calculus can be modelled by a solution to the recursive type equation $X \simeq X \to X$. Similarly, a foundation for concurrency based on linear logic will be of limited value if it only models race-free and deadlock-free processes. Are there extensions that support more general forms of concurrency?

Girard (1987) proposes one such extension, the Mix rule. In our notation, this is written as:

$$\frac{P \vdash \Gamma \quad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ \mathsf{Mix}$$

Mix differs from Cut in that there are *no* channels in common between P and Q, rather than one. Mix is equivalent to provability of the proposition $A \otimes B \multimap A \parr B$ for any A and

https://doi.org/10.1017/S095679681400001X Published online by Cambridge University Press

B. Systems with Mix still do not deadlock, but support concurrent structures that cannot arise under CLL, namely, systems with two components that are independent. An example in Section 3.2 introduced a primitive computation

$$\mathsf{par}_{y,z} \vdash y : 1, z : 1$$

which is equivalent to the Mix rule. Mix is defined in terms of the primitive by setting

$$P \mid Q \stackrel{\text{def}}{=} \nu z.(\nu y.(\mathsf{par}_{y,z} \mid y().P) \mid z().Q)$$

Equivalently, the primitive can be defined in terms of Mix by setting

$$\mathsf{par}_{y,z} \stackrel{\text{def}}{=} y[].0 \mid z[].0$$

Caires *et al.* (2012a) consider two variations of the rules for 1 and $\bot$, the second of which is less restrictive and, surprisingly, derives a rule similar to Mix.

Abramsky *et al.* (1996) propose another extension, the Binary Cut rule (a special case of Multicut). In our notation, this is written as:

$$\frac{P \vdash \Gamma, x : A, y : B \quad Q \vdash \Delta, x : A^{\bot}, y : B^{\bot}}{\nu x{:}A, y{:}B.(P \mid Q) \vdash \Gamma, \Delta}\ \mathsf{BiCut}$$

Binary Cut differs from Cut in that there are *two* channels in common between P and Q, rather than one. Binary Cut is equivalent to provability of the proposition $A \parr B \multimap A \otimes B$ for any A and B. Binary Cut allows one to express systems where communications form a loop and may race or deadlock.

Systems with both Mix and Binary Cut are compact in that from either of $A \otimes B$ and $A \parr B$ one may derive the other. Abramsky *et al.* (1996) provide a translation of full π-calculus into a compact linear system, roughly analogous to the embedding of untyped λ-calculus into typed λ-calculus based on the isomorphism $X \simeq X \to X$. Searching for principled extensions of CP that support the unfettered power of the full π-calculus is a topic for the future work.

Session types have been developed in many directions since being introduced by Honda (1993). Among the most important of these is multi-party session types, introduced by Honda *et al.* (2008) and elaborated by many others. Another topic for the future work is whether the logical foundations introduced by Caires & Pfenning (2010) and further developed here extend to multiparty session types.

As λ-calculus provided foundations for functional programming in the last century, may we hope for this emerging calculus to provide foundations for concurrent programming in the coming century?

Acknowledgments

For comments and discussions, my thanks to Samson Abramsky, Luis Caires, Marco Gaboardi, Simon Gay, Andy Gordon, Marc Hamann, Jiansen He, Kohei Honda, Sam Lindley, Garrett Morris, Luke Ong, Frank Pfenning, Colin Stirling, Michael Stone, Vasco Vasconcelos, Hongseok Yang, Nobuko Yoshida, Stephan Zdancewic, and the anonymous referees. This paper is dedicated to the memory of Kohei Honda, 1959–2012.

References

Abramsky, S. (1993) Computational interpretations of linear logic. *Theor. Comput. Sci.* **111**(1 & 2), 3–57.

Abramsky, S. (1994) Proofs as processes. *Theor. Comput. Sci.* **135**(1), 5–9.

Abramsky, S., Gay, S. J. & Nagarajan, R. (1996) Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School, NATO ASI Series F*, pp. 35–113.

Bellin, G. & Scott, P. J. (1994) On the pi-calculus and linear logic. *Theor. Comput. Sci.* **135**(1), 11–65.

Benton, N. & Wadler, P. (1996) Linear logic, monads and the lambda calculus. In *Logic in Computer Science (LICS)*, pp. 420–431.

Berger, M., Honda, K. & Yoshida, N. (2005) Genericity and the pi-calculus. *Acta Inform.* **42**(2–3), 83–141.

Caires, L., Pérez, J. A., Pfenning, F. & Toninho, B. (2013) Behavioral polymorphism and parametricity in session-based communication. In *European Symposium on Programming (ESOP)*, *Lecture Notes in Computer Science*, vol. 7792. Berlin, Germany: Springer, pp. 330–349.

Caires, L. & Pfenning, F. (2010) Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR 2010)*, Paris, France, pp. 222–236.

Caires, L., Pfenning, F. & Toninho, B. (January 2012a) Towards concurrent type theory. In *Types in Language Design and Implementation (TLDI)*, pp. 1–12.

Caires, L., Pfenning, F. & Toninho, B. (2013) Linear logic propositions as session types. *Mathematical Structures in Computer Science*. To appear in Special Issue on Behavioural Types.

Carbone, M. & Debois, S. (2010) A graphical approach to progress for structured communication in web services. In *Interaction and Concurrency Experience (ICE)*, pp. 13–27.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G. C., Larus, J. R. & Levi, S. (2006) Language support for fast and reliable message-based communication in singularity OS. In *European Conference on Computer Systems (EuroSys)*, pp. 177–190.

Gallier, J. H. (1990) *On Girard's "Candidats de Reducibilité"*. Academic Press, pp. 123–204 .

Gay, S. J. & Hole, M. (2005) Subtyping for session types in the pi calculus. *Acta Inform.*, **42**(2–3), 191–225.

Gay, S. J. & Vasconcelos, V. T. (2010) Linear type theory for asynchronous session types. *J. Funct. Program.* **20**(1), 19–50.

Girard, J.-Y. (1987) Linear logic. *Theor. Comput. Sci.* **50**, 1–102.

Girard, J.-Y. (1991) A new constructive logic: Classical logic. *Math. Struct. Comput. Sci.* **1**(3), 255–296.

Girard, J.-Y., Lafont, Y. & Taylor, P. (1989) *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge, UK: Cambridge University Press.

Herbelin, H. (2005) On the degeneracy of sigma-types in presence of computational classical logic. In *Typed Lambda Calculi and Applications (TLCA)*, pp. 209–220.

Honda, K. (1993) Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR 1993)*, Hildesheim, Germany, pp. 509–523.

Honda, K., Vasconcelos, V. T. & Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming (ESOP)*, pp. 122–138.

Honda, K., Yoshida, N. & Carbone, M. (2008) Multiparty asynchronous session types. In *Symposium on Principles of Programming Languages (POPL 2008)*, pp. 273–284.

Kobayashi, N. (2002) Type systems for concurrent programs. In *Proceedings of the 10th Anniversary Colloquium of UNU/IIST*, LNCS, vol. 2757. Berlin, Germany: Springer-Verlag, pp. 439–453.

Kobayashi, N., Pierce, B. C. & Turner, D. N. (1996) Linearity and the pi-calculus. In *Principles of Programming Languages Symposium (POPL 1996)*, St. Petersburg Beach, FL, pp. 358–371.

Kobayashi, N. & Yonezawa, A. (1993) ACL – A concurrent linear logic programming paradigm. In *International Logic Programming Symposium (ILPS)*, pp. 279–294.

Kobayashi, N. & Yonezawa, A. (1994) Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, LNCS, vol. 907. Berlin, Germany: Springer-Verlag, pp. 137–166.

Kobayashi, N. & Yonezawa, A. (1995) Asynchronous communication model based on linear logic. *Form. Asp. Comput.* **7**(2), 113–149.

Mazurak, K. & Zdancewic, S. (2010) Lolliproc: To concurrency from classical linear logic via Curry–Howard and control. In *International Conference on Functional Programming (ICFP 2010)*, pp. 39–50.

Merro, M. & Sangiorgi, D. (2004) On asynchrony in name-passing calculi. *Math. Struct. Comput. Sci.* **14**(5), 715–767.

Miller, D.. (1992) The pi-calculus as a theory in linear logic: Preliminary results. In *Extensions to Logic Programming*, LNCS, vol. 660. Berlin, Germany: Springer-Verlag, pp. 242–264.

Milner, R., Parrow, J. & Walker, D. (1992) A calculus of mobile processes, i. *Inf. Comput.* **100**(1), 1–40.

Pérez, J., Caires, L., Pfenning, F. & Toninho, B. (2012) Linear logical relations for session-based concurrency. In *European Symposium on Programming (ESOP)*, pp. 539–558.

Pfenning, F., Caires, L. & Toninho, B. (2011) Proof-carrying code in a session-typed process calculus. In *Certified Programs and Proofs (CPP)*, pp. 21–36.

Pierce, B. C. & Sangiorgi, D. (2000) Behavioral equivalence in the polymorphic pi-calculus. *J. ACM* **47**(3), 531–584.

Pierce, B. C. & Turner, D. N. (2000) Pict: A programming language based on the pi-calculus. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Plotkin, G. D., Stirling, C. and Tofte, M. (eds). Cambridge, MA: MIT Press, pp. 455–494. ISBN: 978-0-262-16188-6.

Pitts, A. M. (2011) Structural recursion with locally scoped names. *J. Funct. Program.* **21**(3), 235–286.

Sumii, E. & Kobayashi, N. (1998) A generalized deadlock-free process calculus. (*High-Level Concurrent Languages (HLCL), 1998*). *Electron. Notes Theor. Comput. Sci.* **16**(3), 225–247.

Takeuchi, K., Honda, K. & Kubo, M. (1994) An interaction-based language and its typing system. In *Proceedings of the 6th International PARLE Conference*, Athens, Greece, Halatsis, C., Maritsas, D. G., Philokyprou, G. and Theodoridis, S. (eds), LNCS, vol. 817. Berlin, Germany: Springer-Verlag, pp. 398–413.

Toninho, B., Caires, L. & Pfenning, F. (2012) Functions as session-typed processes. In *Foundations of Software Science and Computation (FoSSaCS)*, pp. 346–360.

Toninho, B., Caires. L. & Pfenning, F. (2013) Higher-order processes, functions, and sessions: A monadic integration. In *22nd European Symposium on Programming (ESOP'13)*, Lecture Notes in Computer Science, vol. 7792 . New York, NY: Springer, pp. 350–369.

Turner, D. N. (1995) *The Polymorphic Pi-Calculus: Theory and Implementation*, PhD thesis, University of Edinburgh, Edinburgh, UK.

Vasconcelos, V. T. (2011) Sessions, from types to programming languages. *Bull. Eur. Assoc. Theor. Comput. Sci.* **103**, 53–73.

Wadler, P. (September 2012) Propositions as sessions. In *International Conference on Functional Programming (ICFP)*, pp. 273–286.

Yoshida, N. & Vasconcelos, V. T. (2007) Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.* **171**(4), 73–93.